J.T. Yao, V.V. Raghvan, G.Y. Wang (Eds.): WSS'04, pp. 45-51, 2004.

Self-Protection of Web Content

Hoi Chan, hychan@us.ibm.com Trieu C. Chieu, , tchieu@us.ibm.com

IBM T.J.Watson Research Center
19 Skyline Drive
Hawthrone NY, 10532

Abstract

In most Internet applications, there is little control on how to protect the data content once it reaches the client. Implementing centralized control for data content delivered to the client at the server side is complicated and requires frequent server and client interaction which may influence user experience negatively. This suggests that data contents embedded with self-protecting functions which run independently on the client side may be desirable. In this paper, we propose an approach utilizing existing browser and agent technology to enable content protection function by using agent embedded in the delivered content. We illustrate this approach by using a web mail application, in which the content of the display page is locked up automatically using embedded functionalities in the html page when no activities are detected for a period of time even when server connection is not available.

Introduction

As the complexity and use of web applications increases, building systems with agent technology[1,2,3] to perform various tasks autonomously becomes increasingly important. Much of the research and development on agents and its related technologies focuses on searching, mining, comparing, negotiating, learning and collaborating. Very little attention has been given to autonomic protection functions[4], especially to protect individualized data content once it reaches the client.

Typically, a server protects data access by disconnecting and logging out a user if no user activities are detected for a period of time. However, this does not prevent the current display page with sensitive information at the client workstation from being read by others when the workstation is unattended. To include functions which can protect or reconfigure the data content requires functionalities beyond what the typical browser can provide. These protection functions can be implemented, to a certain extent, by using a browser plug-in[5] which extends the functionalities of the browser to provide the necessary protection functions. However, in most of the cases, proprietary plug-ins need to be installed explicitly in a browser, and may not be readily available at the client side for the specific data content and applications. In addition, plug-ins in general offer pre-defined and generalized functionalities, it lacks the flexibility to configure itself dynamically to meet individualized needs.

In this report, we describe an approach to use intelligent agents embedded in the delivered content to provide functions specific to the application data or users. This embedded agent approach not only eliminates the need for plug-ins, it also greatly enhances its flexibility to provide functions tailored to specific user needs. In other words, different agents (such as applets[6]) with different protection mechanisms can be dynamically configured and included in the delivered web page based on user information and other criteria.

Data Content Embedded with Agent

In general, for web based application, a browser allows the user to select a link and retrieve another screen of information. The browser itself does not provide a lot of functionalities to manipulate the data. However, Java agent and plug-in technology allow extension of browser functions, and provide Java like functionalities in the browser environment. Therefore, it becomes possible to include a set of functions embedded in the data content which implements the various protecting functions by using agent technology. A Java[TM] agent, such as an applet, is a program written in Java programming language that can be included in a html page, much in the same way an image or text are included. When you use a Java technology-enabled browser[7] to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM).

Typically, user activities and other browser parameters can be accessed through methods in JavaScript[8] in html pages. To enable an applet to access information from the current html pages, we need a mechanism to go beyond the boundary of Java Runtime Environment (JRE) of the applet and connect the applet with the JavaScript in the html pages. This is achieved by using the LiveConnect[9] facility, which is readily provided in most commercial browsers. Basically, this facility provides a netscape.javascript.JSObject class[10], which permits applets to work with JavaScript to enable a way to access the document-object-model (DOM)[11] of an html page.

Based on this facility, we have developed a design to provide a time out protection function embedded in web page. The root of the design is a Java class called *AbstractWebContentAgent* class. It provides a set of common functions to allow communication between Java applet and JavaScript to access the content of the html page. Figure 1 is a list of sample methods in the *AbstractWebContentAgent* class. Figure 2 and 3 show the basic Java applet code to access the html page through the use of JavaScript and LiveConnect facility. The *readContent* method is used to read the DOM of the html page, while the *writeContent* method is used to write data back to the original page. These two methods utilize the JSObject class to access the DOM of the html page.

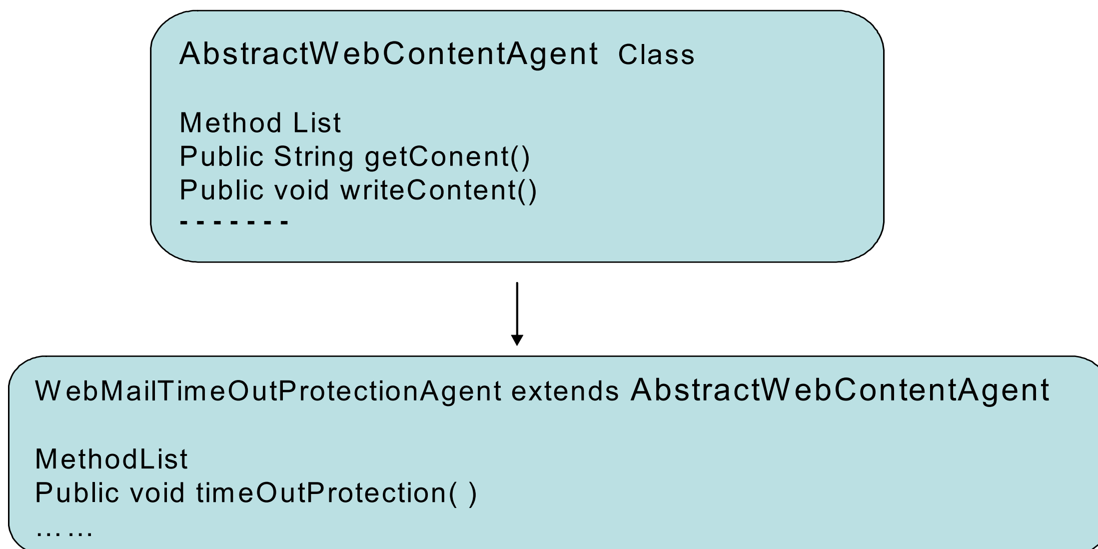

Figure 1. methods provided in the AbstractWebContentAgent class and its extension - WebMailTimeOutProtectionAgent.

```
JSObject thisWindow = JSObject.getWindow(this);
JSObject document = (JSObject) thisWindow.getMember("document");
JSObject myMember = (JSObject) document.getMember("myMember");
// get the member "myRef" as a string
String s = (String) myMember.getMember("myRef");
```

Figure 2. Reading from the html page using JSObject

```
JSObject thisWindow = JSObject.getWindow(this);
JSObject document = (JSObject)thisWindow.getMember("document");
String htmlText = "myText";
args = new Object[ ] {htmlText};
document.call("write", args);
```

Figure 3. Writing to html page using JSObject

Protecting Function for Web Content

The protection scenario involves a company exposing sensitive mail information due to workstations frequently left unattended. The company wants to protect certain sensitive mail pages from displaying in client workstations when no user activity is detected for a certain period of time, even after server connection is disconnected. To achieve this, an autonomic protection function is needed which can react to user idling time. This function will lock out the current display page after a specified period of time if no user activities are detected and the user is required to enter the password again to return to the current page. To allow the mail page to be returned for viewing, the lock-up page should be stored locally and securely in order for the user to retrieve it, even when no connection to server is available.

Based on these requirements, one of the most flexible and cost effective approaches is to embed an agent, such as a Java applet, in selected mail pages. The applet performs the time-out protection function and utilizes the *WebMailTimeOutProtectionAgent* class (WMTOPA). This WMTOPA class extends the *AbstractWebContentAgent* class given above and allows the applet to gain access to the content of web mail page. through the *readContent* and *writeContent* methods.

Another function provided by this applet is to store the mail page locally after a time-out period is reached. To avoid security exposure, we need to encrypt the page before storing. To this end, we have developed a double encryption algorithm which uses a public/private key mechanism[12,13] together with the user password to encrypt and decrypt the mail content. The algorithm for the time-out protection and encryption mechanism at the client browser is illustrated in Fig. 4,5 and 6.

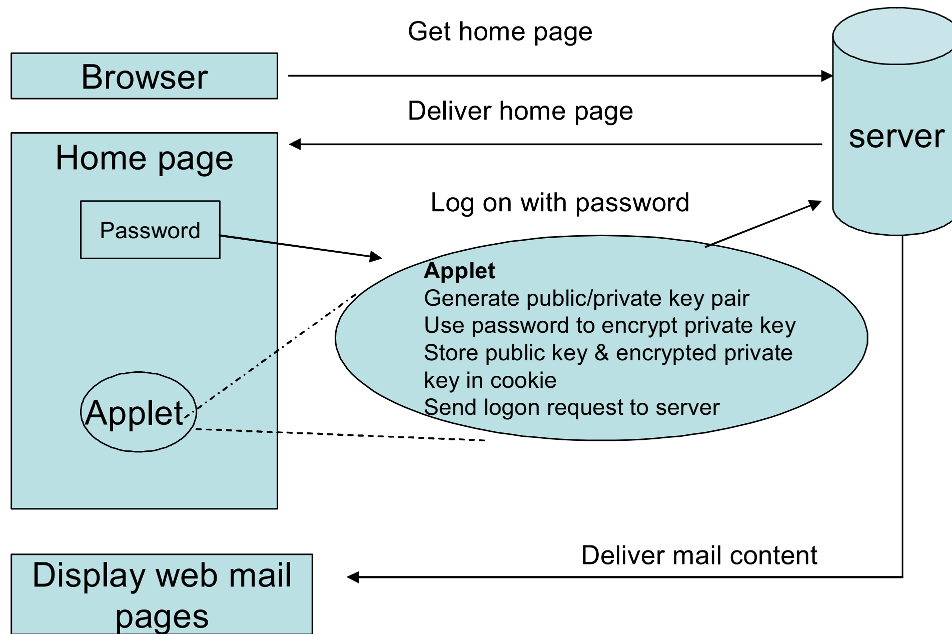

Figure 4. Mechanism to generate self-protecting behavior – locally generate encryption key

The operation sequence illustrated in Figure 4 for generating initial encryption keys is summarized below:

1. User requests mail home page.
2. Server delivers home page to user.
3. User enters a password to login. Applet generates a public/private key pair ($K_{private}$, K_{public}).
4. Applet uses the password entered by the user to encrypt $K_{private}$ to get $K_{encrypted\ private\ key}$.
5. Applet stores K_{public} and $K_{encrypted\ private\ key}$ in session cookie[14].
6. Send login request to the server.
7. Upon successful password verification, server delivers the requested mail page to browser.

The reason for the above operation is to avoid using the user password for future encryption/decryption of the mail page, which requires storing the unencrypted password locally. This may cause a security exposure. To address this issue, we use a double encryption algorithm with a public/private key mechanism, and encrypt the private key using user password during initial logon. By keeping public and encrypted private keys in the session cookie, we can use the public key to encrypt information when needed, and retrieve the encrypted private key for decryption when a user enters the same logon password.

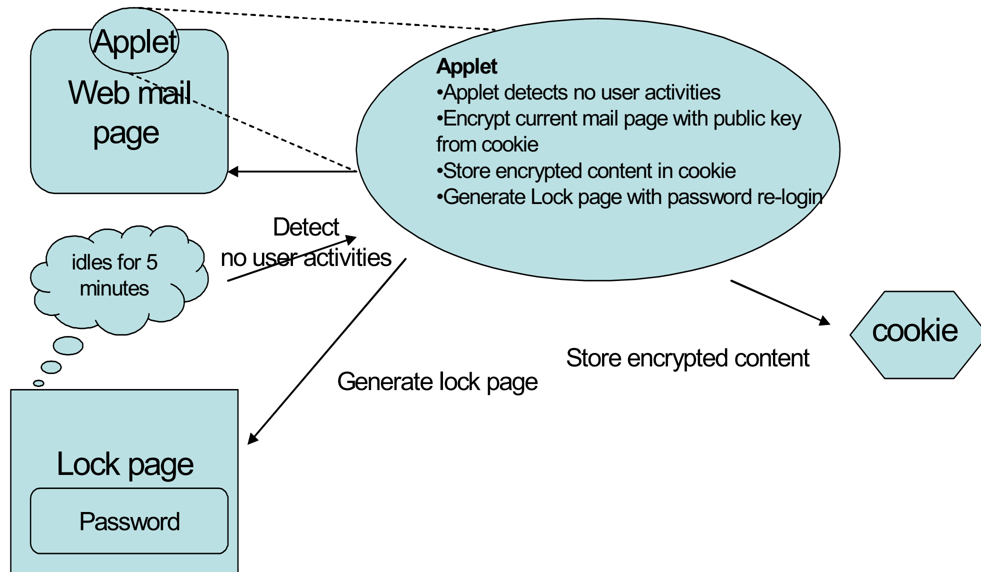

Figure 5. Mechanism to generate self-protecting behavior – locally encrypt and store mail content, display Lock page with re-login

The operation sequence illustrated in Fig. 5 for locking and storing an encrypted mail page locally is summarized below:

1. Mail page is displayed on user's workstation.
2. If no user activities detected within a period of time (e.g. 5 minutes)
3. The current mail page is encrypted with public key K_{public} retrieved from session cookie.
4. The encrypted content of the entire mail page is stored in session cookie
5. Generate a lock page with password unlock prompt

.

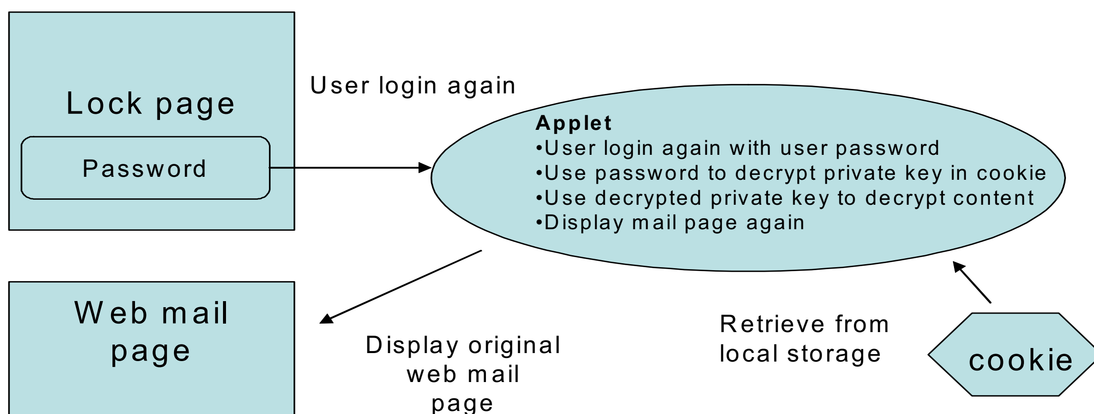

Figure 6. Mechanism to generate self-protecting behavior – locally retrieve, decrypt and display upon successful re-logon with password

The operation sequence illustrated in Figure 6 for retrieving and decrypted the web mail page is summarized below:

1. User enters password to unlock page
2. Applet retrieves $K_{encrypted\ private\ key}$ from session cookie and decrypt it with user password to get $K_{private}$.
3. Applet retrieves encrypted content from session cookie and decrypt it with $K_{private}$
4. Display original web page.

In the above example, we achieved storing the mail content locally and avoid security exposure by encrypting the information using a public/private key mechanism together with the user password. The generation of the encrypted private key $K_{encrypted\ private\ key}$ using the user password guarantees that the original key $K_{private}$ is securely stored locally. The original private key can be readily recovered when a user enters the password to unlock the page.

The important features illustrated by this example are the time-out and security functions that are provided by the attached applet agent in the html page. In general, different applets can be attached to different html pages depending on their data content and specific needs. It is also important to note that information is securely stored locally to satisfy the requirement that the user can still retrieve the locked page even where there is no server connection. This is achieved by using a double encryption technique using the pubic/private keys and the user password as the main entrance key.

Conclusions

In this report, we have described an implementation of an agent embedded in web content which provides a self-protection function for a delivered page without a browser plug-in even after server connection is disconnected. We have also described a double encryption scheme which allows secured mail content to be stored locally at the client side to avoid security exposure. The methodology described in this report provides a convenient way of adding autonomous functions[4] to web data contents at the client side, especially in situations where specialized plug-ins are not available. This approach and concept can be extended to enable other self-managing functions and personalized behaviors for web content using dynamic attachment of agents depending on data content, user information and other environment criteria in a client/server distributed environment. Despite many of its good points, this approach has its drawbacks, namely, its limitation by the size of the applet and the complexities of maintaining a repository of available applets.

References

[1] Intelligent Agents: Theory and Practice 12/2/99 - Mike Wooldridge and Nick Jennings, Intelligent Agents: Theory and Practice, Knowledge Engineering Review, v10n2, June 1995.
[2] Agent-Based Engineering, the Web, and Intelligence – Charles J. Petrie. December 1996 issue of IEEE Expert.
[3] Intelligent Agents in Cyberspace - 1999 AAAI Spring Symposium, http://www.aaai.org/Press/Reports/Symposia/Spring/ss-99-03.html.
[4] Jeff O. Kephart, David M. Chess, "The Vision of Autonomic Computing", Computer Journal, IEEE Computer Society, January 2003 issue
[5] Java Plug-in Component - http ://java.sun.com/j2se/1.4.2/docs/guide/plugin/
[6] Applet Resources - http://java.sun.com/applets/
[7] Java-enabled browsers - http://physics.syr.edu/courses/java/browsers.html
[8] JavaScript to Java Communication – http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/js_java.html
[9] LiveConnect/Plug-in Developer's Guide http://wp.netscape.com/eng/mozilla/3.0/handbook/plugins/
[10] JSObject – http://wp.netscape.com/eng/mozilla/3.0/handbook/plugins/doc/netscape.javascript.JSObject.html
[11] DOM – Document Object Model http://www.w3.org/DOM/

[12] Public Key Cryptography -
http://en.wikipedia.org/wiki/Public_key
[13] Public Key Cryptography
http://www.verisign.com/repository/crptintr.html
[14] Browser Session Cookie
http://www.mach5.com/support/analyzer/manual/html/

Java™ is a trade mark of Sun MicroSystems Inc.