iAuth: An HTTP Authentication Framework Integrated into HTML Forms

Takeru INOUE†a, Yohei KATAYAMA†, Members, Hiroshi SATO†, Nonmember, and Noriyuki TAKAHASHI†, Member

SUMMARY Current Web authentication frameworks have well-known weaknesses. HTTP provides an access authentication framework, but it is rarely used because it lacks presentational control. Forms and cookies, which are most commonly used, have the long-standing privacy issue raised by tracking. URI sessions, which are used in some mobile services like i-mode 1.0, disclose session identifiers unintentionally. This paper proposes iAuth, which integrates better parts of the existing frameworks and fixes their problems; iAuth allows servers to provide log-in forms, and introduces a session header to avoid servers’ tracking and unintentional disclosure. Since iAuth has backward compatibility with the major legacy browsers, developers can freely introduce iAuth into their Web sites or browsers as needed. Experiments confirm its correct operation; an iAuth server is shown to support not only an iAuth client but major legacy browsers. We believe that iAuth will resolve the long-standing issues in Web authentication.

key words: HTTP, HTML, authentication, cookie

1. Introduction

HTTP [5] provides a simple authentication method called the access authentication framework [6]; a server is allowed to ask a client to set a username and a password, which are validated by the server. Some authentication mechanisms, including Basic and Digest authentication schemes, are defined according to this framework. IETF HTTP Bis charter [9] concludes that framework is rarely used despite some degree of support in many implementations because it provides no presentational control for servers; the client shows its own log-in dialog, not the log-in form provided by the server.

Almost all HTTP authentication that involves a human using a Web browser is accomplished through HTML forms and cookies [12], [13]. The server transfers a log-in form to the client, which returns the username and password entered by the user. After validating the username and password, the server stores the username together with a session identifier. The session identifier is set in the cookie of the response, and is automatically copied to successive requests by the client. The server extracts the session identifier from the requests and finds the associated username. Unfortunately, cookies raise privacy concerns due to cookie tracking; if a user happens to enter some private information, her browsing habits in the site may be associated with an individual even if she has not logged in’. Since cookies are opaque to the clients, servers can track clients by setting cookies without being detected.

There are some mobile services that do not support cookies, e.g., i-mode 1.0 [21]. In these services, session identifiers are set as a query string in the requesting URI [3]. We call those sessions whose identifiers are set in URIs URI sessions in this paper. In URI sessions, users are likely to disclose their session identifiers unintentionally, since URIs are usually not considered to be sensitive information. In fact, a bookstore user happened to post a URI that included a session identifier, and her session was hijacked [20].

In this paper, we study client authentication frameworks like HTTP access authentication and HTML-forms and cookies (or -URIs), not authentication mechanisms such as challenge-and-response and public-key signature schemes. Here, we define an authentication framework as an abstraction for the input of credentials on a client and their subsequent transfer between a server and a client; e.g., in the HTTP access authentication framework, credentials are input through a client dialog and transferred in a specified HTTP header, while in HTML form and cookies, credentials are input through a log-in form and transferred in the request body and cookie. Technically, any pair-wise authentication mechanism can be applied to these frameworks, as long as it allows a server and a client to exchange the necessary information in a specified sequence.

This paper focuses on improvement of the authentication frameworks. The problems mentioned above are related to authentication frameworks. They can be categorized into those associated with credential inputs or credential transfer;

- Problem with credential inputs: No presentational control, and
- Problems with credential transfer: Server’s silent track-

†The authors are with NTT Network Innovation Laboratories, NTT Corporation, Yokosuka-shi, 239-0847 Japan.
a) E-mail: takeru.inoue@ieee.org
DOI: 10.1587/transcom.E94.B.466

Copyright © 2011 The Institute of Electronics, Information and Communication Engineers
ing and unintentional disclosure of session identifiers. These problems are caused by the current defective frameworks, and not by the authentication mechanism used. Accordingly, they can be fixed only by improving the frameworks. It is worth noting that credential protection techniques like SSL/TLS provide no solution, since they happen above the encryption layer.

We propose a novel authentication framework, iAuth to solve these problems. Our basic idea is that iAuth combines the better parts of existing authentication frameworks; it borrows its credential input method from HTML forms and cookies, while it takes its credential transfer method from the HTTP access authentication framework. Since these methods were never intended to interwork, we provide the necessary bonds in iAuth.

Our contributions are summarized as follows.

• We propose an authentication framework, iAuth, which provides both better credential input and better credential transfer at the same time. This is our main motivating force as mentioned above.
• We describe how to integrate authentication mechanisms into iAuth, since iAuth does not have its own authentication mechanisms.
• iAuth offers backward compatibility with the major legacy browsers. This implies that developers can incrementally introduce iAuth into their Web sites or browsers.
• We allow iAuth servers to specify the order of implementing multiple mechanisms in the response, one of which is chosen by the client.
• We conduct experiments to verify iAuth operation, by running an iAuth server against an iAuth client as well as the major legacy browsers: Internet Explorer, Firefox, Safari, Opera, and Chrome.
• We discuss roles of cookies beyond session management; one, anonymous sessions, can be provided by iAuth, and the other, storage for site settings, can be replaced by the local storage of HTML5 [8].

In this paper, we focus on the HTTP layer and assume that other layers’ features (e.g., confidentiality protection and server certification) will be applied as needed. We also assume that a client is connecting to an authentic server, and that the server may track the client without asking. The client system is trustable for the user. Credentials are safely shared between the server and the client in advance.

The remainder of this paper is organized as follows. Section 2 reviews the existing authentication frameworks. Section 3 overviews iAuth and Sect. 4 describes details. Section 5 describes the experiments conducted and Sect. 6 discusses migration scenarios and security considerations. Section 7 summarizes related work and finally Sect. 8 concludes with our contributions and future work.

2. Existing Authentication Frameworks

This section describes the existing authentication frame-works and discusses their issues. Section 2.1 describes the HTTP access authentication framework. Section 2.2 shows HTML forms and cookies, and Sect. 2.3 explains URI sessions.

2.1 HTTP Access Authentication Framework

HTTP provides a simple authentication framework called the access authentication framework [6]. Basic and Digest authentication schemes (mechanisms) are defined according to this framework. The user is expected to register his username and password to the server in advance. The server sets the authentication scheme in the WWW-Authenticate header of the response, which is transferred to the client. The client shows a log-in dialog, into which the user enters his username and password, and places the entered information in the Authorization header of the following requests. The server finally validates the username and password in the request. Both headers include an authentication scheme followed by lists of attribute-value pairs, which are defined for each scheme.

Figure 1 shows an example of Basic authentication mechanism. Upon receiving a request for resources in a protected realm, the server responds to the client with a WWW-Authenticate header, which includes the desired authentication scheme (Basic) and the realm attribute. The user enters his username and password into a dialog popped up by the client. The client computes a response value; the username and password are joined by a colon (joe:secret), and then converted into the base64 representation (am9lOnNlY3JldA==). The response value is con-

†An exception to header format is found in Basic authentication; the Authorization header includes a response value without attribute name.
Digest authentication is used only through secure channels, because the base64 representation can be easily converted into plain text.

Digest authentication has a more secure authentication mechanism than Basic, since it transfers a hash value, not the password itself, as shown in Fig. 2. The server responds to the client with a WWW-Authenticate header, which includes the authentication scheme (Digest) and several attributes, such as nonce and opaque, depending on a qop (quality of protection) directive\(^\dagger\). The client computes a response value by hashing the values in the header as well as the username and password entered by the user. The client makes a request with the Authorization header including the response value, some values in the previous response, and the username. The server validates the response value or simply checks the nonce to avoid heavy hash processing (nonce is a session identifier), in identifying the requesting user.

Servers are not allowed to provide log-in forms in the access authentication framework; instead, clients show their own dialog. Even if servers were allowed to do so, clients would not understand the semantics of HTML forms (which element is the username?) and could not compute a response value. Because of this lack of presentational control, the framework is rarely used despite some degree of support in many implementations [9].

2.2 HTML Form Authentication and Cookie Sessions

Almost all HTTP authentication that involves a human using a Web browser is accomplished through HTML forms, with session identifiers contained in cookies. HTML forms provide a large degree of control over presentation, which is imperative for many Web sites. For cookies, most implementations rely on the Netscape specification, which is described loosely in Sect. 10 of RFC 2109 [12], though it was rendered obsolete by RFC 2965 [13]. This framework relies on the password authentication mechanism, and subsequent sessions are maintained by temporal session identifiers.

An example of HTML form authentication and a cookie session is shown in Fig. 3. A user enters his username and password into a log-in form provided by a server. This information is transferred as a query string in the request body, and validated by the server. If authentication succeeds, the server generates a session identifier and stores it together with the username, as shown in the session table in Fig. 3. The session identifier is set in the Set-Cookie header in the response, which is sent to the client. Subsequently, the client is forced to set the session identifier in the Cookie header of all requests sent to the server. The server extracts the session identifier from the requests and finds the associated username.

\(^\dagger\)We omit the qop directive in Fig. 2 for simplicity, though it should be specified to avoid the vulnerability of obsolete specifications [7].
Clients are not required to understand the semantics of the log-in form, since users' inputs are converted into a query string representation without being processed, unlike the access authentication framework. As noted in Sect. 1, cookies, however, have a privacy issue called tracking cookies; a server can track users by setting a session identifier without authentication. The client does not understand whether a session identifier exists in his cookie because cookies are opaque to clients.

2.3 URI Sessions

Session identifiers are set as a query string in the requesting URI in some mobile services that do not support cookies, e.g. i-mode 1.0 [21]. We illustrate this approach in Fig. 4. First, the server authenticates the user by using HTML forms and stores the session identifier in itself, in the same manner as described in the previous subsection. Next, the server guides the client to URIs including the session identifier, by returning a response with a Location (redirection) header and links to the URIs. As a result, the client requests the URIs of this session. The server extracts the session identifier from the requesting URIs, and finds the corresponding username.

URI sessions are easily hijacked if the session URIs are disclosed on a Web site. Unfortunately, URIs are not treated as sensitive information, so this problem is significant. As described in Sect. 1, actually this vulnerability has caused known cases of session hijacking.

3. iAuth: Basic Idea

First, we summarize authentication frameworks and their problems, as discussed in Sect. 2.

- HTTP access authentication framework: Credentials are input through a client dialog and transferred in the Authorization header. Servers are not allowed to provide log-in forms for the input of credentials.
- HTML forms and cookies (and URIs): Credentials are input through a log-in form and transferred in the cookie (or URI). Clients cannot detect server tracking, because the cookie (or URI) is opaque to the clients. In addition, session identifiers can be disclosed unintentionally in URI sessions.

These problems are obviously independent of whatever authentication mechanism is used and can be fixed only by improving the framework.

This analysis leads to the basic idea of iAuth; our authentication framework borrows the better parts from the existing frameworks; it takes its credential input method from HTML forms and cookies, while it borrows its credential transfer method from the HTTP access authentication framework. This idea is very simple, but these methods were not intended to interwork. Even if servers provide log-in forms, clients would not understand the semantics of the HTML forms (which element is the username?) and cannot compute a response value to be transferred.

iAuth allows servers to show the semantics of the log-in form in the WWW-Authenticate header, thereby allowing clients to compute a response value set in the Authorization header. Servers also show the subscheme (authentication mechanism) used in the authentication process. iAuth has backward compatibility with legacy systems, because legacy browsers are allowed to ignore unknown WWW-Authenticate headers and to employ their traditional cookie sessions.

To summarize, iAuth has the following advantages.

- iAuth realizes presentational control by enabling servers to provide log-in forms and convey their semantics to clients.
- iAuth allows clients to detect sessions \(^\dagger\) by introducing new headers. In addition, the unintentional disclosure of session identifiers is avoided, since no session information is contained in URIs.

We illustrate the behavior of iAuth with Basic authentication (subscheme) in Fig. 5. Upon receiving a request for resources in a protected realm, the server responds to the client with a WWW-Authenticate header and a log-in form. The header includes an authentication scheme (iAuth) and some attributes; subscheme is the authentication mechanism used, realm is the same as in the HTTP access authentication framework, and username and password are the corresponding elements’ name in the log-in form. The client draws the log-in form instead of showing its own dialog. The user enters his username and password into the form, and the client extracts the specified elements from the form. The client computes a response value (base64 representation) in the same manner as the original Basic authentication, and sets

\(^\dagger\)iAuth defines no explicit session unlike cookies or URI sessions, but its interaction implicitly acts as a session. We, therefore, call iAuth interactions “sessions” in this paper.
it in the Authorization header in the requests. The server finally validates the username and password in the Authorization in the request. In this way, iAuth gives presentational control and session detection†.

Figure 6 shows how iAuth offers backward compatibility. The legacy client ignores the WWW-Authenticate header with its unknown scheme (iAuth)††, and draws a login form. The user’s inputs are transferred as a query string in the request. The iAuth server validates the query string if the request has no Authorization header. The iAuth server changes its behavior depending on whether an Authorization header exists or not. If no Authorization header is found, the session is maintained by cookies, not iAuth. In this case, clients would have the same privacy concerns due to cookie tracking, as they do in normal cookie sessions.

4. iAuth Details

This section discusses iAuth in detail. Section 4.1 shows header formats. Sections 4.2 and 4.3 describes server and client processing, respectively.

4.1 Header Formats

iAuth uses the WWW-Authenticate response header and the Authorization request header as does the HTTP access authentication framework. These headers include an authentication scheme followed by lists of attribute-value pairs, as described in Sect. 2.1. The authentication scheme is, of course, iAuth. Table 1 presents attributes used in iAuth for all subschemes. As noted in Sect. 2.1, the realm attribute is the name of the protected realm, and the response attribute includes a value computed by the client to certify that client knows the password. The secure attribute is the same as that set in cookies; the client is required to connect to the server through secure channels. The subscheme attribute specifies

†The session can be detected even with cookies if the semantics are shared between a server and a client. However, it seems difficult to standardize cookie attribute names without conflict, since cookies are being used in a variety of ways. We believe the introduction of a new scheme (iAuth) is the most practical solution.

††Client’s behavior upon receiving unknown schemes depends on implementation, since it is not defined in RFC 2617 [6]. We confirmed the behavior in experiments.
Table 1 Attributes for iAuth.

<table>
<thead>
<tr>
<th>Name</th>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required. Name of protection space.</td>
</tr>
<tr>
<td>subscheme</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required. Authentication mechanism.</td>
</tr>
<tr>
<td>secure</td>
<td>WWW-Authenticate</td>
<td>Optional. Secure channel required.</td>
</tr>
<tr>
<td>response</td>
<td>Authorization</td>
<td>Required. Value for certifying password.</td>
</tr>
</tbody>
</table>

Table 2 Attributes for Basic subscheme.

<table>
<thead>
<tr>
<th>Name</th>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>WWW-Authenticate</td>
<td>Required. Name of username element.</td>
</tr>
<tr>
<td>password</td>
<td>WWW-Authenticate</td>
<td>Required. Name of password element.</td>
</tr>
<tr>
<td>form</td>
<td>WWW-Authenticate</td>
<td>Optional. Name of log-in form.</td>
</tr>
</tbody>
</table>

Table 3 Key attributes for Digest subscheme.

<table>
<thead>
<tr>
<th>Name</th>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required. Same as Basic in WWW-Authenticate, username in Authorization.</td>
</tr>
<tr>
<td>password</td>
<td>WWW-Authenticate</td>
<td>Required. Same as Basic.</td>
</tr>
<tr>
<td>form</td>
<td>WWW-Authenticate</td>
<td>Optional. Same as Basic.</td>
</tr>
<tr>
<td>nonce</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required. Session identifier.</td>
</tr>
<tr>
<td>opaque</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required. Realm identifier for each user.</td>
</tr>
<tr>
<td>uri</td>
<td>Authorization</td>
<td>Required. Requesting URI.</td>
</tr>
</tbody>
</table>

the authentication mechanism to be used.

This paper introduces Basic and Digest subschemes, which were originally specified in the HTTP access authentication framework, as described in Sect. 2.1. Servers are allowed to set multiple subschemes separated by a slash where the order of preference matches the naming order, e.g. subscheme="Digest/Basic," in the WWW-Authenticate header (all attributes required for all subschemes listed are expected to be included)†. The client chooses the preferred subscheme and sets it in the Authorization header.

We describe the key attributes defined for Basic and Digest subschemes.

Basic subscheme. Table 2 shows the attributes defined for Basic subscheme. The WWW-Authenticate header must include the username attribute and the password attribute, as well as realm and subscheme. The form attribute is expected to be included if the response contains more than one form. The Authorization header must include the realm, subscheme, and response attributes. The response attribute uses base64 representation as per the original Basic authentication.

Digest subscheme. Table 3 shows key attributes defined for Digest subscheme. The WWW-Authenticate header must include username, password, nonce, and opaque, as well as realm and subscheme. The Authorization header must include username, nonce, opaque, and uri, as well as realm, subscheme, and response. The response attribute has a hash value that is computed as in the original Digest authentication.

4.2 Server Processing

Figure 7 shows a sequence diagram of an iAuth server given a request to a protected realm. If the request has an Authorization header with iAuth scheme listed, the server authenticates the request according to the specified subscheme in the header. The server responds with 200 status code (OK) on success, or 401 status code (Unauthorized) on failure. The failure response should include the reason in a human-readable format.

If the request includes a query string, not an Authorization header, the server executes form authentication and sets a cookie for session management. The following processes are same as iAuth.

If the request has neither an Authorization header nor a query string, the server responds with 401 status code.

The response of 401 status code must include a WWW-Authenticate header and a log-in form. The log-in form must specify POST method, not GET, to prevent the inclusion of a password in the URI. The log-in form is allowed to include some elements other than username and password.

†iAuth does not rely on the mechanism described in RFC 2617 Sect. 1.2 that allows servers to list multiple schemes in a response, since it cannot show the server preference.
Table 4 Attributes for a new subscheme based on the RSA authentication.

<table>
<thead>
<tr>
<th>Name</th>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session identifier</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>username</td>
<td>Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>service name</td>
<td>Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>method name</td>
<td>Authorization</td>
<td>Required. “publickey.”</td>
</tr>
<tr>
<td>boolean</td>
<td>Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>public key algorithm</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>public key blob</td>
<td>WWW-Authenticate and Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>public key</td>
<td>Authorization</td>
<td>Required.</td>
</tr>
<tr>
<td>signature</td>
<td>Authorization</td>
<td>Required.</td>
</tr>
</tbody>
</table>

Fig. 8 Sequence diagram of iAuth client.

4.3 Client Processing

Figure 8 shows a sequence diagram of an iAuth client upon receiving a response. If the response has 401 status code and a WWW-Authenticate header of iAuth scheme, the client makes a request whose Authorization header shows iAuth; it chooses the preferred subscheme and computes a response value according to the subscheme. The client is also required to set a query string in the request body as usual, which may be used by the server. If the WWW-Authenticate header has errors, e.g., invalid values or no required attributes, the client can send just a query string without an Authorization header.

If the client receives a successful response to the authentication request, it can set a log-in icon to inform the user of log-in status, similar to the pad lock icon shown by browsers for secure channel communication (https). It is worth mentioning that the log-in icon cannot be shown in cookie sessions and URI sessions, since they are opaque to clients as described in Sect. 2.

4.4 Introduction of New Subschemes

iAuth allows the introduction of new subschemes (authentication mechanisms), as long as they offer a pair-wise client authentication mechanism. Any new subscheme needs to define a message format and sequence. Any new attributes comprising the format can be defined, if the attribute follows the HTTP header syntax [5] and has no conflict with other subschemes. iAuth has no restriction on message sequence if the sequence follows the server-client model, though the sequence diagrams in Figs. 7 and 8 might become a bit complicated.

We briefly discuss the introduction of RSA authentication, like that used in SSH (i.e., “publickey” method in [22]). RSA authentication introduces several new attributes and requires more round-trips than Basic and Digest authentication. The new attributes can be simply defined as shown in Table 4 (see [22] for detailed descriptions). The session identifier is provided by the WWW-Authenticate header of the server’s 401 response. The client, then, makes a request with all attributes other than a public key and signature, and the server responds with the session identifier, a message number, public key algorithm, and public key blob. The client again sends a request with all attributes other than public key blob, and the server finally responds with the session identifier and a message number after validating the public key and signature. In this way, there is no difficulty in supporting RSA authentication in iAuth, though the protocol details (e.g., key sharing and error handling) are beyond the scope of this paper.

It is worth noting that iAuth does not support authentication mechanisms with third-parties like Kerberos, since iAuth is a pair-wise authentication framework.

5. Experiments

We conducted experiments to check iAuth operation. The iAuth server was implemented by using Catalyst [18], a Web application framework written in Perl. The server supports Digest subscheme as well as Basic subscheme. We implemented the iAuth client on Firefox. This client supports only Basic subscheme. Figure 9 shows the messages captured by LiveHTTPHeaders†. Upon receiving a request to a protected realm, the server responded with 401 status code and the WWW-Authenticate header (Fig. 9(1)). This response specified Digest and Basic subschemes. The user entered username and password into the log-in form provided by the server (Fig. 9(2)). The client chose Basic subscheme and issued a request with Authorization header (Fig. 9(3)). The server found the Authorization header and authenticated the

†LiveHTTPHeaders is a Firefox extension to capture HTTP messages. It is found at http://livehttphheaders.mozdev.org/.
request according to Basic subscheme. In this way, iAuth enables servers to control the log-in forms.

Next, we made some experiments against major browsers presented in Table 5, and i-mode 1.0 browsers running on the mobile phones (N706i and SH-04A). All browsers, except i-mode, ignored the WWW-Authenticate header with its unknown scheme (iAuth), and drew the traditional log-in forms. The user’s inputs were transferred as a query string in the request (Fig. 10). Finding that the query string had no Authorization header, the server authenticated the request and set cookies in response. The experiments showed that iAuth does provide backward compatibility with major browsers. The i-mode browsers, however, showed an error dialog, not a log-in form, because of the WWW-Authenticate header with unknown scheme. This means that i-mode sites must check the UserAgent header to ensure that the WWW-Authenticate header is not sent to legacy i-mode browsers. This issue is not significant because most i-mode sites have already such a mechanism in place.

6. Discussion

We discuss migration scenarios in Sect. 6.1 and security considerations in Sect. 6.2.
Table 5 Browsers used in experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>7.0</td>
</tr>
<tr>
<td>Firefox</td>
<td>3.5.3</td>
</tr>
<tr>
<td>Safari</td>
<td>4.0.3</td>
</tr>
<tr>
<td>Opera</td>
<td>10.0</td>
</tr>
<tr>
<td>Chrome</td>
<td>3.0</td>
</tr>
</tbody>
</table>

6.1 Migration Scenarios

Our goal is to replace defective cookie sessions and URI sessions with iAuth. As shown in the experiments, iAuth has backward compatibility with legacy systems, thereby allowing iAuth servers and clients to be introduced independently. We expect that some Web sites with high-level security needs will be willing to support iAuth without waiting for the support of all major browsers. iAuth extension for Web application frameworks, such as Ruby on Rails and Catalyst, encourages the spread of iAuth servers. Similarly, iAuth extensions for major browsers will also be helpful. It is worth mentioning that iAuth also provides means to promote the use of new authentication mechanisms through the listing of multiple subschemes.

Cookies and URI sessions should be made obsolete. Although they have two roles other than session management, these roles can be supported as follows.

Anonymous session. An example of an anonymous session is seen in the use of shopping carts; shopping carts are assigned to each not-logged-in user anonymously, and the carts are taken over by the users when they log in the site. Cookies and URI sessions support anonymous sessions by issuing session identifiers without authentication. Clients, however, cannot understand the semantics of this session, which causes issues like tracking cookies. iAuth servers are allowed to maintain anonymous sessions by providing a form with hidden elements (a username and a password), in which the server has set random unique values. The values are a kind of session identifier; they are issued at the first visit, and are transferred to the following requests as a username and password in the Authorization header. An example of the iAuth anonymous session is presented in Fig. 11. This message flow has the minimum round-trips for shopping and is almost same as the traditional anonymous ses-
sions, in which session identifiers are issued in cookies or URIs at the first visit and exchanged in the following inter-
actions. iAuth anonymous sessions involve no extra inter-
action over the traditional frameworks. In addition, clients
are allowed to show a log-in icon unlike cookies and URI
sessions, since iAuth behaviors are understood by clients.

Storage for site settings. An example of this storage is seen in saving the favorite font size for a Web site. The size is
usually stored in cookies, but it does not have to be trans-
ferred to the server. It can be stored in the local storage provided by HTML5 and managed by JavaScript.

We believe that many Web sites as well as browsers
would be motivated to switch to iAuth to improve their se-
curity levels. Cookies could be forbidden by default given a
sufficient number of iAuth servers; third party cookies are
already forbidden by default in Safari. It is not easy to
detect URI sessions, but they would be suppressed by public
pressure.

6.2 Security Considerations

iAuth inherits a security issue from the form authentication;
untrusted JavaScript codes must not be included to prevent a
malicious third party from capturing form inputs. Authentic-
tion mechanisms introduced into iAuth might bring their
own security threats; e.g., base64 representation of Basic
subscheme can be easily converted into plain text, and Di-
gest subscheme is vulnerable against dictionary attacks.
Lower layer’s confidentiality protection should be applied if
needed, or more secure authentication mechanisms should
be introduced as shown in Sect. 4.4. Identifying the optimal
mechanism is a future work.

iAuth resolves the issues of servers’ tracking and unin-
tentional disclosure of session identifiers, but we still have
some issues that are independent of authentication and ses-
sion management.

Phishing. Phishing is the criminally fraudulent process of
masquerading as a trustworthy entity to acquire sensitive
information like passwords. Basic and Digest subschemes
have no means to certify servers. Phishing can be resolved
by establishing a secure channel or checking blacklists.

Cross-site request forgery. Cross-site request forgery
transmits unauthorized commands from a user that the Web
site trusts. It is not prevented by the introduction of iAuth,
because iAuth clients keep the log-in state like cookies. To
prevent this vulnerability, servers should embed a one-time
token in edit pages and check it on submission, as before.

Cross site scripting and SQL injection. Cross site script-
ing and SQL injection are vulnerabilities that enable ma-
licious attackers to inject client-side script or SQL com-
mands into Web sites. iAuth has no means to verify users’
inputs, which should be carefully verified by servers as is
done now.

7. Related Work

Reference [15] examined how cookies and browser designs
had evolved due to concerns about informed consent. They
concluded that while cookie technology has improved over
time regarding informed consent, some of the original prob-
lems remain; clients are forced to accept cookies regardless
of the semantics. iAuth enables clients to specify the seman-
tics as described in this paper.

References [19], [23] propose unique methods to detect
tracking cookies; a client sends two requests with and with-
out a cookie, and checks the validity of the cookie by iden-
tifying differences in the responses. However, this check is
easily avoided by slightly changing the responses. We con-
clude that cookies should be replaced by iAuth.

Reference [10] discusses a variety of tracking ap-
proaches including a range of browser cache methods and
inspecting the color of a visited hyperlink. Since it is diffi-
cult for authentication methods to resolve these attacks, they
propose that the same-origin policy should be applied to the
cache and links as well as JavaScript and cookies.

OpenID [17] and OAuth [1] offer the single sign-on and
delegation of access authorities, respectively. While forms
and cookies are used in the current practice, other methods
can be also applied. OpenID and OAuth would be more
secure by introducing iAuth.

SSL/TLS [4] provide credential protection and server
authentication in most of use cases, while iAuth provides
client authentication. Credential protection is of no help
in solving these problems, since they happen above the en-
cryption layer. SSL/TLS can provide client certificate au-
thentication; FirstPass [16] is an example for mobiles. This
method avoids the problems mentioned above, but their use
is limited as far as we know.

8. Conclusions and Future Work

This paper proposed iAuth, which enables servers to provide
log-in forms in the HTTP access authentication framework.
iAuth does not have its own authentication mechanism and
supports any mechanism. It offers backward compatibil-
ity with legacy systems. We made experiments to confirm
its validity, by running an iAuth server against not only an
iAuth client but major legacy browsers. We presented mi-
gration scenarios and discussed security considerations.

We have found no practical issues with iAuth so far,
but technology integration can often cause tricky problems
in general terms. It would be a good idea to introduce an
indirection layer between HTTP and HTML: for example,
new attributes are defined for specifying username and pass-
word elements, instead of the name attribute used in this pa-

1Unlike the traditional anonymous sessions of cookies and
URIs, the anonymous session of Fig. 11 has the overhead of cre-
dential validations (demanded by Basic and Digest authentica-
tion). Though our focus is not on authentication mechanisms as such, we
brieﬂy discuss this overhead. We believe that the simple encoding
 techniques used in Basic authentication, like base64 representa-
tion, generate no significant overhead. The hash functions used in
Digest authentication require heavy computation, but this over-
head can be simply reduced by checking a nonce attribute instead of
validating credentials with hash functions [6].
per. We hope that such details will be discussed in standards groups including IETF httpsbis Charter.

References


Takeru Inoue received the B.E., M.E., and Ph.D. degrees from Kyoto University, Kyoto, Japan, in 1998, 2000, and 2006, respectively. Since joining NTT Network Innovation Laboratories in 2000, he has been engaged in R&D on mobile networking, multicasting, and Web architecture. His research interests are in large-scale communication networks. He received the best paper award from Asia-Pacific Conference on Communications in 2005. He also received the research awards of the IEICE Information Network Group in 2002 and 2005. He is a member of IEEE.

Yohei Katayama received the B.E. and M.E. degrees from Keio University, Kanagawa, Japan, in 2007 and 2009, respectively. Since joining NTT Network Innovation Laboratories in 2009, he has been engaged in R&D on multicast technology. His research interests are in large-scale communication networks. He received the research awards of the IEICE Internet Architecture Group in 2008.

Hiroshi Sato received the B.S. and M.S. degrees in mathematics from Tohoku University, Sendai, Japan, in 1994 and 1996, respectively. In 1996, he joined NTT Communication Science Laboratories, where he engaged in research on natural language processing. He moved to NTT Network Innovation Laboratories in 2007, and has been researching on ubiquitous data management. He won the best paper award of IPSJ National Convention in 1999. He is a member of Information Processing Society of Japan (IPSJ) and Japan Society for Fuzzy Theory and Intelligent Informatics (SOFT).

Noriyuki Takahashi received the B.E. and M.E. degrees in information engineering from Kyoto University, in 1990 and 1992, respectively. He joined NTT laboratories in 1992. He has been in the NTT Network Innovation Labs. since 2001. His current interests include routing mechanisms, adaptive networking systems, and architecture design of future networks. He is a member of IPSJ, ACM, and IEEE.