Efficient Cookie Revocation for Web Authentication

Ruopeng Ye, Agnes Chan, Feng Zhu

College of Computer and Information Science, Northeastern University, Boston, MA, USA

Summary
Many web-based services use persistent cookies to store user authentication information on the disk. In these services, when a web browser connects to the server, it sends the persistent cookies to automate the authentication process so that the user does not need to type in the username or password. However, current web authentication architecture does not have a proper expiration mechanism. As a consequence, a hacker can use an expired cookie to gain unauthorized access to the web services. To fix this problem, we propose two schemes for the web servers to efficiently store and verify cookie state information. We show that these schemes can effectively stop the replay-attack from expired cookies and can be easily implemented.

Key words:
Cookie revocation, Web authentication.

1. Introduction

The HTTP cookie [5] is a text string generated by a web server when responding to a HTTP request from a web browser. It is sent to the browser via the Set-Cookie header in the HTTP response. The cookie is identified by its name as well as the domain and path of the server that generates it. When a browser connects to a web server, it will put all cookies with a matching domain and path in the Cookie header of the HTTP request. Thus although HTTP is a stateless protocol [6], a web server can use cookies to store state information such as registration and authentication, user preferences, and other information in a web browser and can retrieve it at a later time.

Many web sites hold protected content which requires user authentication, such as web mail services, e-commerce services, and online subscription services. These web sites usually require a user to sign in by supplying an account name and password. After the user is authenticated successfully, the web server puts an authentication cookie in the browser so that when the user accesses the protected content the server can retrieve the cookie for authentication verification. It improves usability as the account name and password do not need to be typed in every time an access to some protected content is made.

However, the authentication cookie needs to be revoked and expired in a timely manner, to guard against impersonation. Currently, the expiration is usually done by removing the authentication cookie from the web browser. In the simple case, if the cookie does not have the EXPIRES attribute, it will be stored in the memory only, so by closing the web browser the cookie will be deleted and the authentication will expire. In the general case, many web sites provide a logout button to terminate the authentication. When the button is clicked, the browser sends a logout request and the server responds with a Set-Cookie header to set the authentication cookie to a null string. In both cases, the authentication expiration process solely depends on the browser, the web server does not have any record of which authentication cookie has been expired. If an authentication cookie is stolen and replayed by a malicious user, who can then gain access to the protected web content. Thus the current authentication expiration mechanism gives users a false sense of security.

The goal of our work is to provide authentication expiration methods that can be depended on. Our main contribution is to devise efficient methods for web servers to maintain cookie state for web authentication sessions. We propose two schemes, the “Simple scheme” and the “M/K scheme”. They are both efficient, and guarantee immediate authentication expiration, which cannot be offered by current web authentication system that uses cookies. Our methods require the server to maintain a small amount of information per user so that once the user requests an authentication expiration, the server can update and remember the validity state of the authentication cookie. We have implemented and tested both authentication expiration mechanisms on Apache [13] web server, the result shows that the performance impact on the web server due to extra processing is insignificant.

2. Related Work

Using persistent cookies (these are the cookies which have the EXPIRES attribute explicitly specified, and have not yet expired) to store authentication information is known to be vulnerable to attacks. Usually such cookies have their SECURE attributes marked so that they will be transmitted only in the encrypted communication channel using HTTPS. But because most browsers save persistent

Manuscript received January 5, 2007
Manuscript revised January 25, 2007

cookies in a plain text file which is not protected at all, it can be easily stolen and replayed [17]. Kormann and Rubin [1] provided a session hijacking attack where an attacker can impersonate a user by stealing the authentication cookie from the victim's computer. Fu et al. [3] showed a poisoned cookie attack where an attacker can gain unauthorized access to a web site by modifying a cookie which is not cryptographically protected. In [4], a cross-site scripting attack is discussed, where a malicious web server is able to steal a user's cookie from the web browser. To prevent replay attack using stolen cookies, in [19], Liu et al. proposed to use the SSL session key as a keying material for the web server, to generate a key to protect the cookies. This solution is only suitable for protecting session cookies, it cannot be used for persistent cookies, as the SSL session key is changed over multiple sessions

Noting all these problems, the authors in [3] recommended that authenticator should not be stored in persistent cookie. We argue that an authentication cookie does have its advantage and should be used and not be discarded entirely. First, authentication cookie reduces the number of manual sign-in and hence makes the web site more user friendly. Second, as the HTTP is stateless, cookie is currently the most widely deployed mechanism for maintaining client state [3]. Some largest web mail services, such as Gmail (including the mobile Gmail) and Hotmail (see Figure 1) are using authentication cookies to improve the usability. In Kerberos [11], the authenticator (authentication ticket) is encrypted by a user-specific master key and stored in a locally trusted workstation. The problem with authentication cookies is that most web browsers do not provide strong encryption protection to the cookie file, resulting in authentication cookies easily stolen and replayed [18].

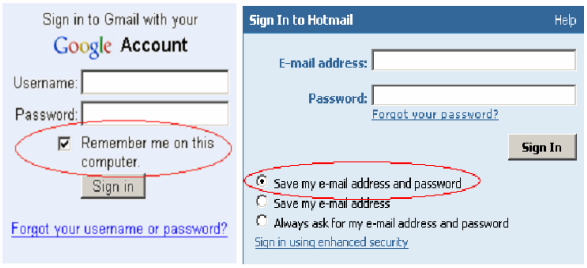

Fig. 1 Gmail and Hotmail Sign-in Page.

In [14], Schneier pointed out the importance of terminating authentication and argued that users should be given the opportunity to delete their usernames and passwords and terminate the accounts. In [15], Stubblebine presented a general method for specifying authentication revocation with any desired degree of immediacy. The work is based on public-key cryptosystem. It defines several properties that a revocation service should provide, such as being definite and fail-safe; that is, revocation should remain effective under unreliable communication. ASP.NET [16] uses a session object for state management, this can be used to for authentication revocation. However, this technology is platform dependent, and cannot be used in all web servers. In this paper, we propose two efficient authentication expiration mechanisms that operate on any web servers. Our schemes are very lightweight — the first one takes a constant space, the second uses approximately one bit for each cookie. We show that both schemes are resilient to replay attacks.

The remainder of the paper is organized as follows. In section 3 we describe a commonly used web authentication architecture. Base on this architecture, we show a replay attack on authentication expiration in section 4. In section 5 we present two authentication expiration methods that are definite and immune to the replay attack described in section 4. The security properties of these schemes will also be discussed in section 5. The implementation and performance results of our schemes will be presented in section 6. We conclude the paper in section 7.

3. Web Authentication Architecture

The web authentication architecture we describe here is derived from the authentication framework which is defined by HTTP protocol [7]. It consists of a web authentication server, a web content server, and a client web browser, as shown in Figure 2. This is a very flexible authentication architecture, where the authentication server and the content server can run either jointly or separately. For example, in web servers such as Apache [13], where the authentication server is run as a module (mod_auth) of the web content server, the authentication and the content servers run jointly. On the other hand, in single sign-on systems such as Passport [10], the servers run separately.

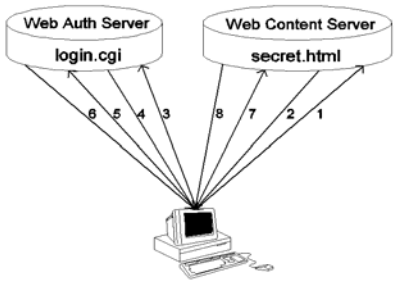

Fig. 2 Web Authentication Architecture (Fresh Login).

The web content server can define some of its documents as protected (this is done by putting the documents under a

protected realm). For instance, in Figure 2 the secret.html is a protected document. When a browser sends a request to the content server to retrieve a protected document, it needs to present a credential (for example, a persistent authentication cookie generated by the authentication server) to be authenticated by the content server. In the case where the content server and authentication server are running jointly, this authentication is usually done with the help from the authentication server (for example, in Apache, the authentication is done by *mod_auth*). If the authentication is successful, the content server will return the document to the browser. If it fails, a 401 unauthorized response will be sent back to the browser. To protect the authentication credential from eavesdropping attack, the communication between the browser and the servers is transmitted through encrypted channels protected by HTTPS protocol.

If the browser does not have an authentication credential, the content server will redirect the browser to the authentication server. This is called a fresh sign-in and can be explained in details in Figure 2. Initially, when the browser requests secret.html (step 1), it has no authentication credential; so the content server redirects it to the authentication server (step 2). The browser follows the redirection (step 3) and gets a user authentication page (for example, the *login.cgi* in Figure 2) from the authentication server (step 4), which asks the user to type in the account name and password for authentication. The authentication information is then sent back to the authentication server (step 5). If the authentication is successful, the authentication server will send an authentication credential to the browser and redirect the browser (step 6) back to the content server. The browser follows the redirection (step 7). After checking the validity of the credential, the content server will then provide the protected document (step 8).

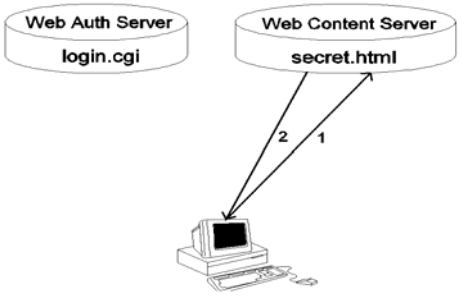

Fig. 3 Web Authentication Architecture (Re-Login/Auto Login).

The above procedure will be much simplified if the browser already has the authentication credential. This is called an auto-login (see Figure 3). The browser sends a request and an authenticator to the content server to get the secret.html document (step 1). The content server checks the authenticator; if successful, it sends back the requested document to the browser (step 2).

To protect the authenticator from being stolen from the browser, an authentication expiration process is executed when the user logs out of the server. The authentication expiration process is depicted in Figure 4. Assume that the secret.html provides a logout button, which links to a page named logout.html. When the user clicks the logout button, the browser sends a request for logout.html (step 1). The content server verifies the authenticator in the request and responds with the document if the authentication is successful (step 2). The header of this response will instruct the browser to delete the authentication credential from the browser to expire the authentication. The user will know the expiration process is finished when the logout.html is received and shown on the browser.

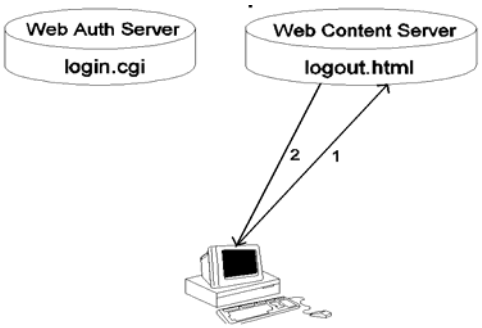

Fig. 4 Web Authentication Architecture (Logout).

4. Replay Attack

As seen from Figure 4, the authentication expiration is not reliable — the content server does not know whether the authentication credential is successfully deleted or not. This makes the authentication expiration mechanism susceptible to replay attack.

To launch a replay attack, first the attacker needs to have a copy of valid authenticator (authentication cookie). Although the communication channels between the browser and the servers (authentication server and content server) are protected by HTTPS, there are still several ways through which an attacker can steal the authentication cookie from a user. One method is to copy the authentication cookie from the cookie manager of the web browser. Another method is to copy the cookie database file from the file system. Because in most browsers the cookie database file is not encrypted and is stored in a known location, this method can be used by both malicious hackers and computer virus. Cross-site

scripting attack [4] is yet another way to steal the authentication cookie.

Once the attacker has a copy of the authentication cookie, he can put it to a browser either through the cookie manager or by overwriting the cookie database file. When connecting to the content server, the authentication cookie will allow the attacker to impersonate the victim user in an auto-login. Note that this attack works even after the victim user clicks the logout button. In this case, the authentication cookie in the victim user's browser is deleted, but the stolen one remains in the attacker's browser, and there is no way that the content server can find out about this. Imagine that a user has a browser window logged in to view some protected document, then the user leaves the computer terminal without locking the screen. When he returns, he realizes the risk that someone else might have stolen the authentication cookie, so he clicks the logout button trying to invalidate any authentication material that is associated with the previous session. However, due to the faulty authentication expiration mechanism, the stolen cookie remains valid. It is this fact that violates the definite property of the current web authentication expiration mechanism.

One way to mitigate this problem is through limiting the lifetime of the authenticators [3]. The length of the lifetime of an authenticator has been a debated topic since its introduction in 2001. In [12], Kohavi and Parekh recommended that the lifetime value should be at least 60 minutes for e-commerce sites. A previous version of Gmail (see Figure 5) set the lifetime of the authenticator to two weeks. If the lifetime of the authenticator is too short, it can cause problems for some applications (for example, loss of shopping cart). In Kerberos [11], the maximum lifetime of an authentication ticket in V4 is about 21 hours, this is increased to virtually unlimited in V5. However, no matter how long or short the lifetime of the authenticator is, there is always a vulnerability window in which the replay attack can be launched successfully.

Fig. 5 Previous Gmail Sign-in Page (2005).

We tested the replay attack on several webmail systems, such as Hotmail and Gmail. These systems allow authentication cookies to be stored on a disk, which makes it simple to carry out the attack. First we launched a fresh sign-in and saved a copy of the cookie database file. After clicking the logout button and receiving the logout confirmation, we confirmed that the authentication cookie was deleted from the browser by connecting to the webmail sites, and noticing that we were forced to do new fresh sign-in. Then we closed the browser and overwrote the cookie database file with the saved copy. When we started the browser and returned to the webmail sites, the saved copy of the authentication cookie enabled us to auto-login to the mailbox. We repeated the test multiple times; the result showed that the replay attack would work as long as it was launched within the vulnerability window.

It is at the webmail systems that we first found that the web authentication expiration mechanism is susceptible to the replay attack. We also conducted an online search and found that many other web sites also offer the option to remember user's password. Combined with the fact that these web sites use cookies to store authentication information, we can infer that these web sites are susceptible to the replay attack described above.

5. Solution

The replay attack works because when the browser sends a logout request to the content server with a valid authentication cookie, the server does not register the state information of the cookie to indicate its expiration. Hence if such a cookie shows up again in a replay attack, the content server will still regard it as a valid authentication cookie. To prevent such an attack, our solution is to devise an efficient scheme for the web server to record the authentication expiration state information of the cookies.

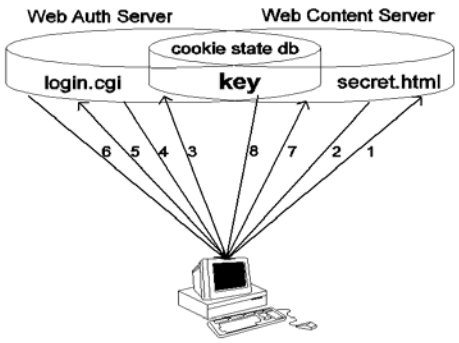

Fig. 6 Authentication Architecture with Stateful Content Server.

We will use the web authentication architecture discussed in section 3 to explain our authentication expiration

schemes. For simplicity, we assume that the authentication server and the content server are running jointly, we will discuss the case for single sign-on in section 5.4.

The system diagram is depicted in Figure 6. The cookie state database is managed by both the content server and the authentication server, each user has an access control entry in the database. There is a secret key *Key* which is used by both servers to generate and verify authentication cookies. We present two different schemes below.

5.1 Simple Scheme

In the simple scheme, the access control entry (see Figure 7) records the most recent time when a logout request was made by the user (initially it is set to *0* to indicate that no logout request was made). When the user executes a fresh sign-in, the authentication server generates an authentication cookie (see Figure 8) as follows. It constructs a message *Msg* consisting of the user name, the IP address of the user's computer, and the current time. Then it computes a keyed MAC (for example, HMAC) of *Msg* using the server secret key *Key*. The authentication cookie is the concatenation of the MAC code and the message *Msg*. The authentication server sends this cookie back to the browser and redirects it to the content server.

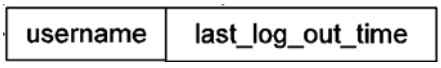

Fig. 7 Access Control Entry in Simple Scheme.

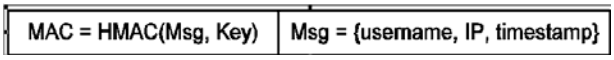

Fig. 8 Authentication Cookie in Simple Scheme.

When the browser requests a protected document from the content server, the content server first validates the cookie using the MAC code, then it compares the timestamp of the cookie with the recorded logout time in the access control entry. If the cookie timestamp is more recent than the recorded time, then the content server returns the protected document to the browser.

When the user clicks a logout button to sign out of an authentication session, the browser sends a request to the content server for a logout.html (see Figure 4). The content server first validates the cookie with the MAC code; if successful then it compares the cookie timestamp with the recorded time in the access control entry. If the cookie timestamp is more recent, the content server will update the access control entry timestamp by setting it to the cookie timestamp.

In short, if a user has signed out using an authentication cookie with timestamp t, all cookies with timestamp $t' \leq t$ (A timestamp is the time in seconds since 00:00:00 UTC, January 1, 1970, so a larger value means more recent.) are considered expired.

5.2 *M/K* scheme

The simple scheme described above may not perform well in parallel sessions. For instance, if a user initiates parallel sessions from multiple computers, then signing out from one computer may interfere with sessions initiated from the other computers. One might use a separate control entry, or a separate state object (in ASP.NET [16]), to manage the cookie states for parallel sessions. But for web mail system where there are a very large number of users, such solutions do not scale well and may lead to denial-of-service attack. To solve this problem, we propose a *M/K* scheme, which allows the webmail server to keep track of a maximum of m authentication cookies within k days. Other time units can be selected in place of day as well, for simplicity reason we will use day in the discussion that follows. Our solution uses bounded space and is updated automatically every day, which alleviates the threat from potential denial-of-service attack. This scheme is very lightweight, in total, ($m + k \log m$) bits are used to mange the m most recent cookies over the last k days.

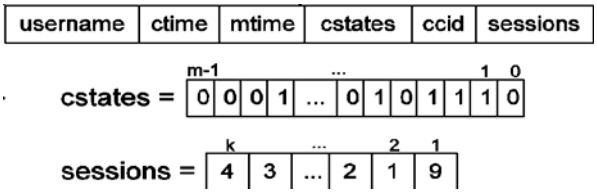

Fig. 9 Access Control Entry in *M/K* Scheme.

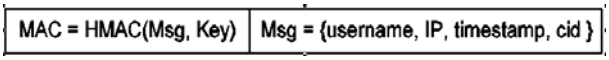

Fig. 10 Authentication Cookie in *M/K* Scheme.

In the *M/K* scheme, the access control entry is constructed as shown in Figure 9. It records the user name, the timestamp when the entry is created (*ctime*), the most recent time the entry is updated (*mtime*), the cookie states vector (*cstates*), the current cookie id (*ccid*), and the session counters (sessions). The cookie states vector (*cstates*) is a m-bit array, each bit records the expiration state of a cookie (*1* for active, *0* for expired), so in total it can keep a log for m authentication cookies. The current cookie id (*ccid*) is an integer in the range of *0* to *m-1*, it represents the cookie id to be used next by the authentication server for cookie generation. The session counters (sessions) is a k-cell array, it counts the number

of authentication cookie generation on a daily basis over a *k*-day period. When the access control entry of a user is initially generated, every field except for the user name will be set to *0*.

When the user executes the very first fresh sign-in, the authentication server sets both the *ctime* and *mtime* to current time *t*, sets *ccid* to *0*, sets the first bit of cookie state vector to *1* (*cstates[ccid] = 1*), and adds *1* to the first cell in the session counters (*sessions[1] = 1*), as shown in Figure 11. It then constructs an authentication cookie (see Figure 10) in the following manner. First it builds a message *Msg* consisting of the user name, the IP address of the user's computer, the current time (*t*), and the cookie id (*cid = ccid*). Then it computes a keyed MAC of *Msg* using the server secret key *Key*. The authentication cookie is the concatenation of the MAC code and the message *Msg*. The authentication server then sends the authentication cookie to the browser and redirects it to the content server.

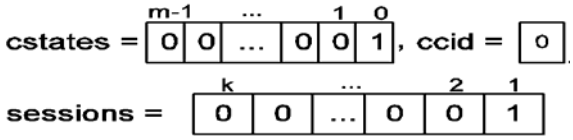

Fig. 11 Access Control Entry in the very first fresh sign-in.

When the user executes a fresh sign-in at other times, the authentication server first sums up the *k* entries in the cookie session counters to determine whether the total number of sessions has exceeded the maximum cookies limit (*m*). The session counters vector is implemented as a shift register which is shifted one cell to the right every day since creation. To do the shifting, the authentication server first computes the days between the current time *t* and the creation time *ctime* and let this number be *cdays*. If *cdays* < *k* then no shifting is necessary. If *cdays* ≥ *k* the server computes the days between *t* and the last modification time *mtime* and let this number be *mdays*, it then shifts the session counters vector *mdays* to the right.

If the sum of the cookie session counters *j* ≥ *m* then the sign-in request is rejected (only *m* fresh sign-in sessions are allowed within *k* days); if *j* < *m*, then the sign-in request can be accommodated. The authentication server first sets *ccid = ccid + 1 mod m*, it also sets the last modification time to be the current time (*mtime = t*), and sets the $ccid^{th}$ bit in the cookie states to *1* (*cstates[ccid] = 1*). Then it adds *1* to the cell in the session counters that corresponds to the current date. This cell can be found by using the current time, the creation timestamp (*ctime*), and the last modification timestamp (*mtime*) in the following way. First computes the days between the current time *t* and *ctime* and let this number be *cdays*. If *cdays* < *k*, the $(cdays + 1)^{th}$ cell (*sessions[cdays + 1]*) is returned; if *cdays* ≥ *k*, the k^{th} cell (*sessions[k]*) is returned. Finally, the authentication server generates the authentication cookie as described in the previous case where the user executes the very first fresh sign-in, sends it to the browser, and redirects the browser to the content server. An example of the result of this procedure can be seen in Figure 12.

When the browser requests a protected document from the content server, the content server first validates the cookie using the MAC code. If successful then it compares the timestamp of the cookie with the current time. The cookie has to be issued in the last *k* days in order to be accepted, otherwise the request is rejected and a fresh sign-in is required. The content server then gets the id of the cookie (*cid*) and looks up the cid^{th} bit in the cookie states vector (*cstates*), if the bit is *0* (which means this authentication cookie has been expired) then the request will be rejected. If the bit is *1*, the content server will return the protected document to the browser.

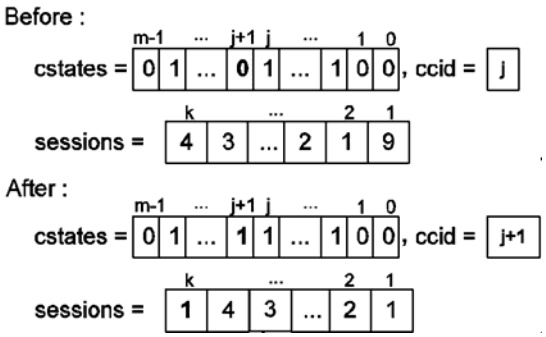

Fig. 12 Access Control Entry in a fresh sign-in.

When the browser sends a sign-out request to the content server, the content server first validates the cookie with the MAC code and verifies that it is issued in the last *k* days. If successful the content server reads the id of the cookie (*cid*) and looks up the cid^{th} bit in the cookie states vector (*cstates*) and sets it to *0*. This prevents the cookie from being replayed. Finally the content server completes the sign-out process by sending the logout.html to the browser.

By using the *M/K* scheme, the cookie session counters (*sessions*) guarantees that a maximum of *m* fresh sign-in sessions can be initiated within *k* days. During the *k* days, the *m*-bit cookie states vector (*cstates*) records the valid state of each cookie (*1* or *0*). The parameters *m* and *k* can be adjusted according to the security requirement. For example, in our implementation discussed in section 6, we set *k = 14* and *m = 128*, which allows an authentication cookie to remain valid for at most *14* days, and a total of *128* fresh sign-in sessions are allowed.

5.3 Security Analysis

The authentication cookie in our schemes is protected from being tampered via a non-malleable MAC. The secret key used to generate the MAC code is known to the authentication server and the content server only. Candidate algorithms include keyed hashes such as HMAC-MD5 and HMAC-SHA1 [9]. To protect the authentication cookie from eavesdropping attack, the *SECURE* attribute of the cookie must be set so that SSL tunneling is used to transmit the cookie between the servers and the browser. Our schemes protects from replay attacks that exploit the weakness of authentication expiration mechanism, however, it does not protect against replay attacks in general. To alleviate the threat from replay attacks which do not depend on the authentication expiration, we have put the client's IP address in the authentication cookie. Although this does not solve the problem completely (for instance, attacker from the same segment of a LAN will have the same IP address), it does make it more difficult to launch the replay attack.

To show that our authentication expiration schemes cannot be exploited for replay attacks, we first consider the simple scheme. Suppose the attacker gets an authentication cookie (with timestamp t') which is associated with an authentication session that has already been expired, then the content server must have updated the timestamp t such that $t \geq t'$. Therefore when the cookie is replayed, the content server will reject the cookie.

For the security of the *M/K* scheme, we will prove the following two theorems.

Theorem 1. In *M/K* scheme, any fresh sign-in will not overwrite the cookie states vector entries (in the *cstates* array) that associate with authentication sessions in the last *k* days.

Proof. If the sign-in is the very first one, there is no previous authentication session, so it will not overwrite the access control entry, this is trivially true.

If the sign-in takes place at other times, then $\sum_{i=1}^{k} sessions[i] < m$, the sum of the cookie session counters is less than *m*; otherwise sign-in is rejected. Therefore there is at least one entry in the cookie states vector that is not associated with any of the authentication sessions in the last *k* days. Next we prove that *cstates[ccid + 1 mod m]* is an unassociated entry that is not associated with any of the authentication sessions in the last *k* days.

Because the only way to update *cstates* array pointer *ccid* is through *ccid = ccid + 1 mod m*, which takes place every time a fresh sign-in occurs, it follows that all the entries in the cookie states vector that are associated with authentication sessions in the last *k* days must be adjacent to each other circularly to the left (see Figure 13).

Therefore, if *cstates[ccid + 1 mod m]* is associated with an authentication session in the last *k* days, then *cstates[ccid + 2 mod m], cstates[ccid + 3 mod m], …, cstates[ccid + m mod m]* (which is *cstates[ccid]*) are all associated with authentication sessions in the last *k* days. This contradicts with the fact that there is at lease one entry in the cookie states vector that is not associated with any of the authentication sessions in the last *k* days. Thus we prove that *cstates[ccid + 1 mod m]* must be an unassociated entry that is not associated with any of the authentication sessions in the last *k* days.

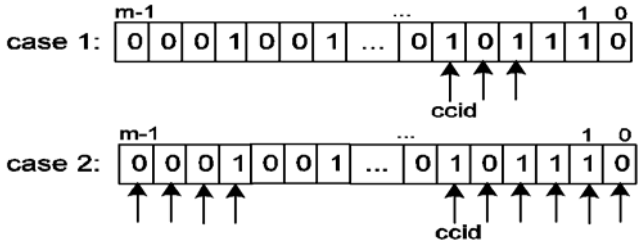

Fig. 13 All entries in cookie states vector ***associated*** with authentication sessions in the last k days are adjacent to each other circularly to the left. ***Associated*** entry is indicated with an arrow.

When processing a fresh sign-in, *cstates[ccid + 1 mod m]* is the only entry in the cookie states vector that is modified, thus we prove that any fresh sign-in will not overwrite any valid cookie states vector (*cstates*) entry that associates with authentication sessions in the last *k* days. □

Theorem 2. In *M/K* scheme, any expired authentication cookie will be rejected..

Proof. In *M/K* schemes, there are two cases in which an authentication cookie is considered expired. First, the cookie had been issued for more than *k* days. Second, the cookie was issued in the last *k* days and was explicitly expired by a sign-out event.

In the first case, the cookie will be rejected because the content server only accepts cookies issued within the last *k* days (by inspecting the cookie timestamp). In the second case, if the cookie has been expired in the last *k* days, the bit associated with this cookie must have been set to *0*. And from Theorem 1, this bit is not modified by any fresh sign-in (only fresh sign-in sets the bits in cookie states

vector to *1*). Therefore, the content server will reject the cookie because its associated bit in the states vector is *0*. □

5.4 Deployability

Both of our schemes require changes only at the server side in the web authentication architecture; they are transparent to the client. This makes the schemes easy to deploy. The secret key used to provide non-malleability for the authentication cookies is kept at the server only, thus the server can re-key at its will. After the secret key is changed, auto-login will fail and the users will be forced to go through fresh sign-in before auto-login can be resumed. In the *M/K* scheme, the server can save a copy of the old key at a re-key event. The old key is saved for *k* days so that when an auto sign-in request is made, both the new key and the old key will be used to validate the authentication cookie, this will help minimize the re-key impact on the users.

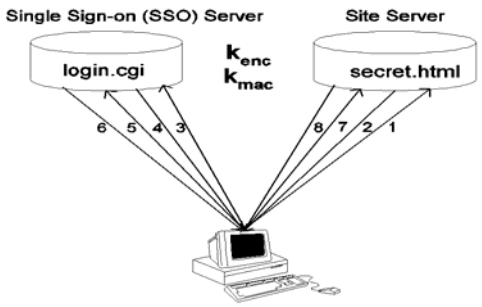

Fig. 14 Single Sign-On Authentication Architecture.

Our schemes can be deployed easily in the common web authentication architecture described in section 3. In section 5.1 and 5.2, we explain how the simple scheme and *M/K* scheme work using an example of the web authentication architecture where the authentication server and the content server run jointly. Here we will describe how the schemes can be deployed in an architecture where the servers run separately. Such an example can be seen mostly in a single sign-on system.

An example of the single sign-on (SSO) system is shown in Figure 14. The SSO server provides authentication service for multiple site servers, it shares a site specific encryption key k_{enc} and message authentication key k_{mac} with each of the site servers. The site server delegates all fresh sign-in to the SSO server for authentication, and the SSO server returns a site specific ticket (encrypted by k_{enc} and integrity protected by k_{mac}) to the browser, which then submits this ticket to the site server for authentication purpose. Because each of the SSO server and the site server has its own authentication credential for auto-login, and the credential is stored as an authentication cookie, therefore our schemes need to be deployed in both the SSO server and the site server to prevent replay attacks on either server.

Figure 15 shows the deployment on a single sign-on architecture. The SSO server has a cookie state database, and a secret key k_{sso} which is used to generate message authentication code (MAC) for SSO server cookies. The site server has its own cookie state database as well, and a secret key k_s to generate MAC code for site server cookies.

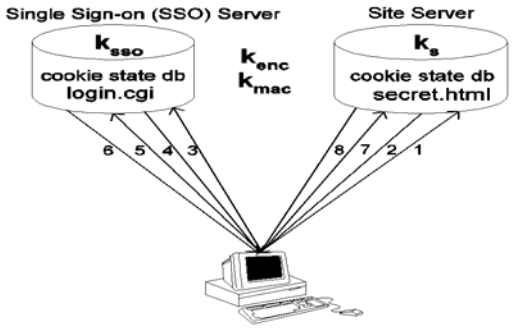

Fig. 15 Single Sign-On Authentication Architecture (with cookie state database).

In a fresh sign-in, the browser requests secret.html from the site server (step 1) and is redirected to the SSO server for authentication (step 2 and 3). The SSO server sets up the access control entry in the cookie state database for the user and returns the *login.cgi* (step 4). The user submits the account name and password for authentication verification (step 5). If the authentication verification is successful, the SSO server updates the access control entry and sends the SSO authentication cookie to the browser. The SSO server also returns a site specific ticket (encrypted by k_{enc} and integrity protected by k_{mac}) through URL redirection (step 6). The ticket has to be sent in the URL because the SSO server and the site server run in different domains, the SSO server can not use cookie to relay the ticket to the site server. The browser then follows the redirection and submits the ticket to the site server (step 7). Upon receiving the ticket, the site server sets up the user's access control entry in its cookie state database, updates the access control entry and generates a site-server authentication cookie. Finally it sends the authentication cookie and the secret.html document to the browser (step 8). After the cookie state databases are set up, the browser can make auto-login and logout in the SSO system as described in section 5.1 and 5.2.

After receiving the SSO server authentication cookie, the user can start single sign-in sessions to other site servers (see Figure 16). Initially the browser requests secret2.html from the site server 2 (step 1) and is redirected to the SSO

server for authentication (step 2). The browser follows the redirection (step 3) and sends the SSO authentication cookie to the SSO server. The SSO server checks the authenticator and returns a ticket for site server 2 (encrypted by k'_{enc} and integrity protected by k'_{mac}) through URL redirection to the browser (step 4). The browser then follows the redirection and submits the ticket to the site server 2 (step 5). Upon receiving the ticket the site server 2 sets up the user's access control entry in its cookie state database, it then updates the access control entry and generates an authentication cookie. Finally it sends the authentication cookie and the secret2.html document to the browser (step 6).

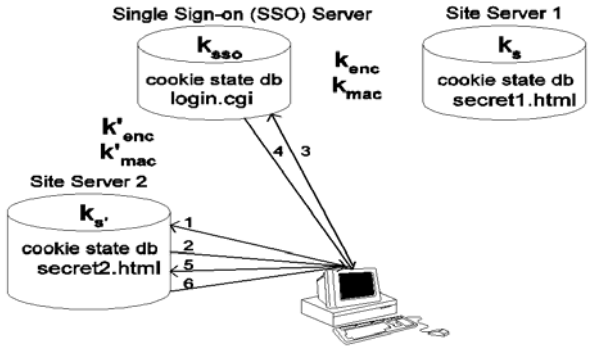

Fig. 16 Single Sign-On Authentication Architecture (new SSO session).

6. Implementation and Performance

We implemented both the simple scheme and the *M/K* scheme on Apache [13] web server. The authentication module was implemented using an open source Apache authentication module *mod_auth_tkt* [8], we used HMAC-MD5 (*128*-bit key) to generate the MAC code of the cookie. We tested our server implementation on a *2.6*GHz Celeron machine with *256*MB of RAM which is running Linux (Fedora Core *4* based on kernel *2.6.11*), Apache (HTTPD v*2.0.55*), and *mod_auth_tkt* (v*2.0.0b6*). We used Lynx (v*2.8.5*) web browser to do the automated testing, as the text based browser is easily integrated into our test script, the client machine is a *377*MHz Pentium III machine with *128*MB of RAM running the same Linux as the server. The client and server are connected via a *100*Mbps link in the same LAN segment.

For each user, we implemented the access control entry discussed in section 5 as a separate file, so that when multiple users are making fresh sign-in requests concurrently, the synchronization access to the access control file does not become a bottleneck of the system performance. In the access control entry, the timestamp takes *4* bytes. For the *M/K* scheme, we chose $m = 128$ and $k = 14$ in our experiment, so the cookie states vector (*cstates*) takes *16* bytes, the cookie session counters (*sessions*) takes *14* bytes, and the current cookie id pointer (*ccid*) takes *1* byte. In fact, both m and k can be set according to the system requirements, once set, the *cstates* vector takes m bits, the *sessions* array takes $k \log m$ bits, and the *ccid* pointer takes $\log m$ bits.

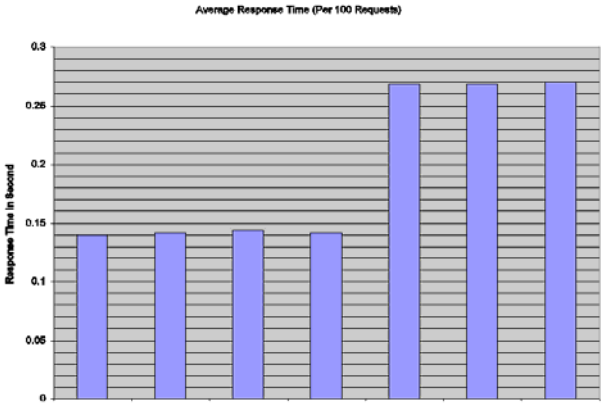

Fig. 17 Average Response Time per 100 Requests.

To study the server performance, we first compare the latency of the fresh sign-in with that of the auto login. The latency is defined as the delay experienced by the browser from the time it sends the request to the time when it receives the response. We ran *1000* trials of the experiment (see Figure 17). Each experiment consists of *100* requests of secret.html under each of the following test scenarios. ***No auth*** is plain HTTPD without authentication. ***Authtkt*** is the HTTPD with the original mod_auth_tkt authentication that does not store cookie state information on the server (this is used as the baseline to compare the performance of our schemes). ***M/K*** and ***Simple*** are the HTTPD with our modified versions of *mod_auth_tkt* which implement the *Simple* and *M/K* schemes.

Figure 17 shows that a request with an auto login takes about the same time (*0.14* second per *100* requests) as a normal request without authentication. While a fresh sign-in takes about twice as much time as it takes to do an auto login. There is no significant difference in the latency between the original mod_auth_tkt authentication and either of our schemes.

Next we compare the concurrency performance of both of our schemes, the HTTPD without authentication, and the HTTPD with original *mod_auth_tkt* module. We launched concurrent connections to the web server under each of the above four configurations, increasing the number of

connections by *10* from *10* to *120* at each run, and measured the time it takes for each run to finish. Figure 18 shows that under all four scenarios the latency increases linearly. While the HTTPD without authentication takes the least time, the performance for the other three implementations is about the same.

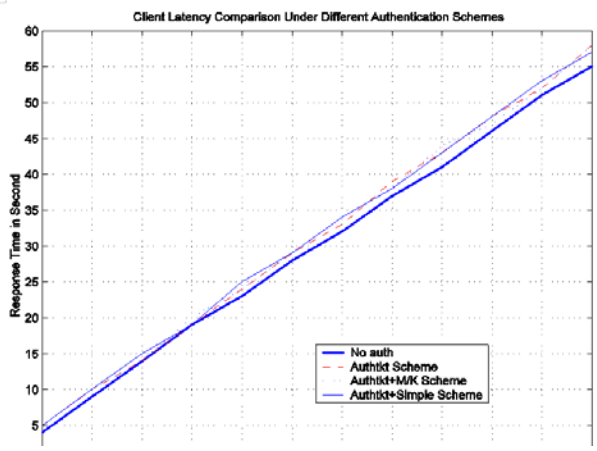

Fig. 18 Average Concurrency Latency Comparison.

7. Conclusion

In this paper, we have studied the current web authentication architecture and pointed out an oversight in the design of its authentication expiration method, which makes it susceptible to replay attacks using stolen cookies from unreliable browsers. We proposed two authentication expiration methods that can thwart this type of attack. They are both effective and efficient, requiring minimal storage on the server to store small amount of information per user. Our experiment showed that neither of the solutions would degrade the server performance. The source code of our implementation can be downloaded at http://www.ccs.neu.
edu/home/robbieye/authexp.tgz.

8. Acknowledgment

We would like to thank Simson Garfinkel and all the anonymous reviewers for comments and suggestions on a preliminary version of the paper.

References

[1] David Kormann, Aviel Rubin, Risks of the Passport Single Signon Protocol, Computer Networks, Elsevier Science Press, volume 33, pages 51-58, 2000.

[2] Charlie Kaufman, Radia Perlman, Mike Speciner, Network Security, Private Communication in a Public World, Prentice Hall, 2002.

[3] Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster, Dos and Don'ts of Client Authentication on the Web, Proceedings of the 10th USENIX Security Symposium, August, 2001.

[4] Malicious HTML Tags Embedded in Client Web Requests, CERT® Advisory CA-2000-02. http://www.cert.org/advisories/CA-2000-02.html.

[5] Netscape: Persistent Client State HTTP Cookies Preliminary Specification, http://wp.netscape.com/newsref/std/cookie_spec.html.

[6] Hypertext Transfer Protocol, http://www.w3.org/Protocols/HTTP.

[7] HTTP Authentication: Basic and Digest Access Authentication, http://www.faqs.org/rfcs/rfc2617.html.

[8] Apache mod_auth_tkt, http://www.openfusion.com.au/labs/mod_auth_tkt/.

[9] HMAC: Keyed-Hashing for Message Authentication, http://www.faqs.org/rfcs/rfc2104.html.

[10] Microsoft: Microsoft .NET Passport Review Guide, http://www.passport.net/.

[11] Kerberos: The Network Authentication Protocol, http://web.mit.edu/kerberos/www/.

[12] Ron Kohavi, Rajesh Parekh, Ten Supplementary Analyses to Improve E-commerce Web Sites, 5th Workshop on Knowledge Discovery in the Web, WebKDD 2003.

[13] Apache HTTP Server Project, http://httpd.apache.org/.

[14] Bruce Schneier, Authentication and Expiration. IEEE Security & Privacy, February, 2005.

[15] Stuart G. Stubblebine, Recent-Secure Authentication: Enforcing Revocation in Distributed Systems, IEEE Symposium on Security and Privacy, 1995.

[16] Fritz Onion, Essential ASP.NET With Examples in C#, Addison-Wesley, 1st edition, 2003.

[17] Joon S. Park, Ravi Sandhu, Secure Cookies on the Web, IEEE Internet Computing, Vol 4, No. 4, July/August 2000.

[18] V. Khu-smith, Chris Mitchell, Enhancing the Security of Cookies, ICICS 2001, LNCS 2288, pp. 132-145, 2002

[19] Alex X. Liu, Jason M. Kovacs, Chin-Tser Huang, Mohamed G. Gouda, A Secure Cookie Protocol, Proceedings of the 14th IEEE International Conference on Computer Communications and Networks, 2005.