Computer Networks 56 (2012) 1723–1730

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

A secure cookie scheme

Alex X. Liu [a,*], Jason M. Kovacs [b,1], Mohamed G. Gouda [c,2]

[a] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266, USA
[b] Exis Web Solutions[3]
[c] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-0233, USA

ARTICLE INFO

Article history:
Received 13 July 2010
Received in revised form 15 October 2011
Accepted 17 January 2012
Available online 31 January 2012

Keywords:
Web cookie
Web security
HTTP authentication

ABSTRACT

Cookies are the primary means for web applications to authenticate HTTP requests and to maintain client states. Many web applications (such as those for electronic commerce) demand a secure cookie scheme. Such a scheme needs to provide the following four services: authentication, confidentiality, integrity, and anti-replay. Several secure cookie schemes have been proposed in previous literature; however, none of them are completely satisfactory. In this paper, we propose a secure cookie scheme that is effective, efficient, and easy to deploy. In terms of effectiveness, our scheme provides all of the above four security services. In terms of efficiency, our scheme does not involve any database lookup or public key cryptography. In terms of deployability, our scheme can be easily deployed on existing web services, and it does not require any change to the Internet cookie specification. We implemented our secure cookie scheme using PHP and conducted experiments. The experimental results show that our scheme is very efficient on both the client side and the server side.

A notable adoption of our scheme in industry is that our cookie scheme has been used by Wordpress since version 2.4. Wordpress is a widely used open source content management system.

1. Introduction

The widely used HTTP (Hypertext Transfer Protocol) works in a request-response fashion. First, a client sends a request (which either asks for a file or invokes a program) to a server. Second, the server processes the request and sends back a response to the client. After this, the connection between the client and the server is dropped and forgotten. HTTP is stateless in that an HTTP server treats each request independently of any previous requests. However, many web applications built on top of HTTP need to be stateful. For example, most online shopping applications need to keep track of the shopping carts of their clients.

Web applications often use cookies to maintain state. A cookie is a piece of information that records the state of a client. When a server needs to remember some state information for a client, the server creates a cookie that contains the state information and sends the cookie to the client. The client then stores the cookie either in memory or on a hard disk. The client later attaches the cookie to every subsequent request to the server.

Many web applications (such as electronic commerce) demand a secure cookie scheme. A secure cookie scheme that runs between a client and a server needs to provide the following four services: authentication, confidentiality, integrity, and anti-replay.

1. *Authentication*: A secure cookie scheme should allow the server to verify that the client has been authenticated

* Corresponding author.
E-mail addresses: alexliu@cse.msu.edu (A.X. Liu), jason@exisweb.com (J.M. Kovacs), mgouda@nsf.gov, gouda@cs.utexas.edu (M.G. Gouda).

[1] The work of Jason M. Kovacs was conducted while he was an undergraduate student of The University of Texas at Austin.

[2] Mohamed G. Gouda is currently a Program Manager at the National Science Foundation.

[3] www.exisweb.com.

1389-1286/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2012.01.013

within a certain time period. Moreover, no client should be able to forge a valid cookie.

In secure web applications, a typical session between a client and a server consists of two phases. The first phase is called the *login phase* and the second phase is called the *subsequent-requests phase*.

(a) *Login phase*: In this phase, the client and the server mutually authenticate each other. On one hand, the client authenticates the server using the server's PKI (Public Key Infrastructure) Certificate after they establish an SSL (Secure Sockets Layer) connection. On the other hand, the server authenticates the client using the client's user name and password, and sends a secure cookie (which is also called an "authentication token" or an "authenticator" in previous literature) to the client.

(b) *Subsequent-requests phase*: In this phase, the client sends the secure cookie along with every request to the server; the server verifies whether the cookie is valid, and if it is, services the request.

2. *Confidentiality*: The contents of a secure cookie is intended only for the server to read. There are two levels of confidentiality that a secure cookie scheme may provide: low-level confidentiality and high-level confidentiality.

(a) *Low-level confidentiality*: A secure cookie scheme with low-level confidentiality prevents any parties except the server and the client from reading the contents of a cookie. To achieve low-level confidentiality, a secure cookie scheme usually runs on top of SSL. Note that SSL encrypts every message between the client and the server using a session key that only the client and the server know. In this paper, we assume that any secure cookie scheme runs on top of SSL.

(b) *High-level confidentiality*: A secure cookie scheme with high-level confidentiality prevents any parties except the server from reading the sensitive information within a cookie that the server does not want to reveal to the client [10]. For example, the cookie's contents may contain some client information such as their internal rating or credit score, which the server may not want the client to be aware of.

Different web applications may require different levels of confidentiality. Therefore, a secure cookie scheme should be able to configured to support both low-level confidentiality and high-level confidentiality.

3. *Integrity*: A secure cookie scheme should allow a server to detect whether a cookie has been modified.
4. *Anti-replay*: In the case that an attacker replays a stolen cookie, a secure cookie scheme should be able to detect that the cookie is invalid. Otherwise, the attacker would be authenticated as the client that the replayed cookie was issued to.

In designing a secure cookie scheme, besides the above security requirements, we also need to consider the issues of efficiency and deployability. As for efficiency concerns, a secure cookie scheme should avoid requiring a server to do database lookups in verifying a cookie, and should avoid public key cryptography. Note that database lookups dramatically slow down the speed that a server takes to verify a cookie. As for deployability concerns, a secure cookie scheme should avoid requiring a client to possess a public key and a private key, which is currently impractical to assume.

Several cookie schemes have been proposed [5,10,16,3]; however, none of these schemes are completely satisfactory. The cookie scheme proposed by Fu et al., has three limitations: (1) it does not have a mechanism for providing high-level confidentiality, (2) it is vulnerable to cookie replay attacks, and (3) it does not provide mechanisms for key updating. The three authentication mechanisms of the cookie scheme proposed by Park and Sandhu are either ineffective or difficult to deploy [10]. The cookie scheme by Blundo et al. [3] and that proposed by Xu et al. [16] are inefficient because they require database lookups in verifying a cookie.

In this paper, we propose a secure cookie scheme that is effective, efficient, and easy to deploy. In terms of effectiveness, our secure cookie scheme provides all of the above four security services. In terms of efficiency, our secure cookie scheme does not involve any database lookup or public key cryptography. In terms of deployability, our secure cookie scheme can be easily deployed on existing web servers, and it does not require any change to the current Internet cookie specification [7].

The rest of this paper proceeds as follows. In Section 2, we present our secure cookie scheme in detail. In Section 4, we discuss the implementation of our secure cookie scheme and its performance. In Section 5, we review and examine existing cookie schemes. We give concluding remarks in Section 6.

2. Secure cookie scheme

The state-of-the-art secure cookie schemes was described by Fu et al. in their seminal paper [5]. In this section, we first examine this scheme, which we refer as *Fu's cookie scheme*. We show that this scheme has three major limitations, and we give a solution to each of them. Finally, we present our secure cookie scheme. The notations used in this section are listed in the following table.

$\|$	Separator
$HMAC(m,k)$	Keyed-Hash Message Authentication Code of message m using key k
sk	Server Key
$(m)_k$	Encryption of message m using key k

The keyed-hash message authentication codes used in this paper are assumed to be *verifiable* and *non-malleable*: given a message m and a key k, it is computationally efficient to compute $HMAC(m,k)$; however, given $HMAC(m,k)$, it is computationally infeasible to compute the message m and the key k. Examples of such keyed-hash message authentication codes are HMAC-MD5 and HMAC-SHA1 [6,2,11,4].

The server key (i.e., sk) of a server is a secret key that only the server knows.

2.1. Fu's cookie scheme

Fu's cookie scheme is shown in Fig. 1.

In this cookie scheme, a secure cookie that is issued by a server to a client consists of the following four subfields within the cookie value field of the HTTP cookie specification.

1. *User Name*: The value of this field uniquely identifies a user.
2. *Expiration Time*: The value of this field indicates when this cookie will expire and the server should reject it from a client. The granularity of this value needs to be small enough for the server to distinguish the time difference of any two cookies.
3. *Data*: The value of this field can be anything that the server wants to remember for the client when the cookie is created. The state information, such as the contents of an online shopping cart or the entry of the state table for the client on the web server, of the communication between the client and the server is usually stored in this field.
4. Keyed-Hash Message Authentication Code (HMAC): The value of this field is the keyed-hash message authentication code of the above three fields using the server key.

Note that separator | can be implemented in many ways as long as we can parse each field correctly. One way is to use a character that is not allowed to be used in any fields, if such a character exists, to be the separator. Another way is to prepend each field with a byte (or more) that indicates the length of the field. We should avoid implementing | as string concatenation because it may cause incorrect parsing.

Using Fu's cookie scheme as an example, we next examine the communication process between a client *C* and a server *S*.

1. Client *C* initiates an SSL connection with server *S*. Client *C* authenticates server *S* using the PKI certificate of *S*. Note that all subsequent messages between *C* and *S* are encrypted by the SSL session key.
2. Client *C* sends its user name and password to server *S*. Server *S* authenticates client *C* by searching a database that contains the user name and hashed password of every client. Up to now, client *C* and server *S* have mutual authenticated each other.
3. Server *S* creates a cookie according to Fig. 1, and sends this cookie back to client *C*.
4. In every subsequent HTTP request sent from *C* to *S*, this cookie is attached. Whenever *S* receives an HTTP request together with a cookie, *S* verifies the cookie by the following two steps:
   (a) Verify expiration time: Server *S* checks whether the cookie has expired by examining the cookie's expiration time and the server's current time.
   (b) Verify keyed-hash message authentication code: Server *S* checks whether the cookie has been modified by recomputing the keyed-hash message authentication code for the cookie. If the result matches the keyed-hash message authentication code in the cookie, then server *S* believes that the cookie is indeed created by *S*.

If the cookie passes both verifications, server *S* services the HTTP request. Otherwise, server *S* denies the HTTP request and asks client *C* to send its user name and password again.

$$\text{user name}|\text{expiration time}|\text{data}|\text{HMAC}(\text{user name}|\text{expiration time}|\text{data}, sk)$$

Fig. 1. Fu's cookie scheme.

Fu's cookie scheme has three major security limitations. First, it does not have a mechanism for providing high-level confidentiality. Second, it is vulnerable to cookie replay attacks. Third, it does not provide mechanisms for key updating. Next, we detail these three limitations and give an efficient and secure solution to each of them.

2.2. Limitation I: cookie confidentiality

We have discussed that some web applications need high-level confidentiality for their cookies. To provide high-level confidentiality, a secure cookie scheme should encrypt the data field of each cookie. Now the question is this: which key should a server use for this encryption?.

Fu's scheme does not provide an answer to this question. There is only one key involved in Fu's scheme, namely the server key. One straightforward solution is to use this server key to encrypt the data field of every cookie; however, this solution is not secure. An attacker can register as a legitimate client with a server and then gather a large number of cookies issued by the server. If the data fields of all of these cookies are encrypted by the same server key, the attacker could possibly discover this key using cryptanalysis. Although such cryptanalysis is hard, it is prudent to avoid any such possibility.

The cookie scheme proposed by Xu et al. [16] manages cookie specific encryption keys as follows. A server maintains a database that stores the user name and the encryption key of every client. When a server creates a cookie for a client, the server generates a new random key for encrypting the cookie and replaces the old encryption key associated with the client in the database with this new key. When a server receives an encrypted cookie from a client, the server uses the user name of the client to search in the database for the corresponding encryption key. This solution has two major disadvantages. First, it is highly inefficient because of the overhead of database lookups. Second, when the old encryption key of a client is deleted, all the cookies that are encrypted by the old encryption key become invalid. This could be very destructive. A client may attach the same encrypted cookie to more than one request to a server. The server may create a new cookie and a new encryption key for the client after the server receives the first request, which results all the other requests with the same cookie being denied by the server.

The cookie scheme proposed by Park and Sandhu manages cookie specific encryption keys in a different way: when a server creates a cookie of high-level confidentiality, the server generates a random key, encrypts the key with the server's public key, and stores the encrypted key in the cookie itself rather than a database on the server side [10]. The downside of this solution is the verification of every cookie involves public key cryptography, which makes the cookie scheme complex and inefficient.

Our solution to this problem is simple and efficient. We propose to use HMAC (user name|expiration time, sk) as the encryption key. This solution has the following three good properties. First, the encryption key is unique for each different cookie because of the user name and expiration time. Note that whenever a new cookie is created, a new expiration time is included in the cookie. Second, the encryption key is unforgeable because the server key is kept secret. Third, the encryption key of each cookie does not require any storage on the server side or within the cookie, rather, it is computed by a server dynamically.

2.3. Limitation II: replay attacks

Fu's cookie scheme is vulnerable to replay attacks, which could be launched in the following two steps. The first step is to steal a cookie that a server issued to another client. An attacker may have several ways to steal a cookie from someone else. For example, if a client stores a cookie in their hard disk, an attacker may steal it using Trojans, worms, or viruses. In the second step of a replay attack, the attacker initiates an SSL connection with the server and replays a stolen cookie that has not yet expired. Consequently, the server incorrectly authenticates the attacker as the spoofed client, and allows the attacker to access the spoofed client's account.

To counter replay attacks, we propose to add the SSL session ID into the keyed-hash message authentication code of a cookie, i.e., to use HMAC (user name|expiration name|data|ession ID, *sk*) as the keyed-hash message authentication code of each cookie. SSL session ID is a random number generated by the server to identify a session with a client. By including SSL session ID in the keyed-hash message authentication code of a cookie, the cookie becomes session specific. Even if an attacker steals a cookie, he cannot successfully replay it because the SSL session ID that the server creates for the attacker is different from the SSL session ID that the server creates for the victim client.

In the preliminary version of this protocol, we proposed to add SSL session key into the keyed-hash message authentication code of a cookie [8]. There are two main advantages to use SSL session IDs, instead of SSL session keys. First, using SSL session IDs leads to one client authentication per SSL session while using SSL session keys leads to one client authentication per SSL connection. Note that one SSL session corresponds to multiple SSL connections and one SSL connection corresponds to one SSL session. In SSL, a session is used to describe an ongoing relationship between a client and a server. A server allows a client to have multiple connections based on the same SSL session. The session ID is a random number generated by the server to uniquely identify a session with a client and it is sent to the client in the SSL handshake protocol. Second, in terms of implementation effort, SSL session IDs are easier to obtain than SSL session keys. There are built in functions in many web programming languages to obtain SSL session IDs. For example, in javex.net.ssl package, the function `getID()` returns the SSL session ID. In SSL, a session, identified by a session ID, is relatively long-lived and it can drive many connections between the client and the server. Servers can expire or preserve client sessions at will within a server-defined timeout period.

To our best knowledge, except the preliminary version of this paper, neither the cookie schemes proposed in prior literature (such as Fu's scheme [5]) nor the cookie schemes used in industry (such as Microsoft ASP.NET authentication scheme [9]) uses SSL session information in forming cookies.

2.4. Limitation III: key updating

Using Fu's cookie scheme, a server uses the same server key in computing the HMAC of every cookie. This server key needs to be periodically changed due to the potential of attackers obtaining the server key by volume attacks or bruthe force key attacks, as stated by Fu et al. [5]. However, periodically changing the server key is problematic. Each time the server key is changed, the server needs to use both the original key and the new key to verify cookies for a certain period. Let t denote the time that the server key is changed from k to k', and Δ denote the largest expiration time that the server has issued for a cookie before time t. Thus, in the time interval $[t, t+\Delta]$, the server may receive legitimate cookies that are created using either the old key k or the new key k'; therefore, if the cookie has not expired and the cookie failed to be verified by the new key k', the server has to use the old key k to verify the cookie. Clearly, verifying cookies twice using two different keys is inefficient.

In our cookie scheme, we propose to use the encryption key, namely HMAC (user name|expiration time, sk), as the key in computing the keyed-hash message authentication code of each cookie. This solution has the following three good properties. First, as discussed in Section 2.2, this key is unique for each new cookie because of the user name and expiration time. Second, this key is unforgeable because of the server key. Third, because this key is the same as the encryption, the computation of HMAC (user name|expiration time, sk) only needs to be done once. Thus, our scheme essentially eliminates the need of updating the key for computing HMAC.

2.5. Our secure cookie scheme

Our secure cookie scheme is shown in Fig. 2. Recall that $(\text{data})_k$ denotes the encryption of the data using key k, and *sk* denotes the server key of a server.

Note that all of the four fields of user name, expiration time, $(\text{data})_k$, and HMAC (user name|expiration time|data|session ID, k) are within the cookie value field of the HTTP cookie specification [7]. The two fields of user name and expiration time are in plain text because the server needs

user name|expiration time|$(\text{data})_k$
|HMAC(user name|expiration time|data|session ID, k)
where k=HMAC(user name|expiration time, sk)

Fig. 2. Our secure cookie scheme.

user name|expiration time
|HMAC(user name|expiration time, k)
where k=HMAC(user name|expiration time, sk)

Fig. 4. Our secure cookie scheme adopted by Wordpress.

to use them to compute HMAC (user name|expiration time, *sk*). Note that the field HMAC (user name|expiration time|data|session ID, *k*) is used by our scheme to provide authentication, integrity, and anti-replay.

Our secure cookie scheme can be configured to provide either high-level or low-level confidentiality. The one shown in Fig. 2 provides high-level confidentiality. If low-level cookie confidentiality is desired, one can simply leave the data field unencrypted, i.e., replace $(\text{data})_k$ by the data in plain text.

The process for verifying a cookie created using our scheme is shown in Figs. 3 and 4. Note that if FALSE is returned, the cookie is deemed invalid and the client must login in again using their user name and password.

2.6. Security analysis

Next, we prove that our cookie scheme is secure. First, our cookie scheme achieves authentication. A cookie created using our scheme can be used as an authentication token because no one can forge a cookie without knowing the server key *sk*, which is only known to the server. Note that HMAC is a one-way collision resistant hash function. Second, our cookie scheme achieves high-level confidentiality. No one can obtain the key *k* for decrypting $(data)_k$ without knowing the server key *sk*. Third, our cookie scheme is secure against replaying attacks. Even an attacker steals a cookie, the attacker cannot establish an SSL session with the same SSL session ID with the server because each SSL session ID is uniquely generated by a server; thus, the stolen cookie in one SSL session is invalid in another SSL session as the session IDs are different. Fourth, for the user who receives a cookie from a server, from the hash HMAC (user name|expiration time|data|session ID, *k*), they cannot infer any information about the data and the server key *k* because HMAC is a one-way collision resistant hash function. In other words, the user will not be able to choose a user name and even data that will allow them to infer the server key *k*. Note that SSL session ID is sent in clear and there is no need to keep it confidential. Last, our cookie scheme is secure against volume attacks because the data encryption key is used only in one SSL session.

2.7. User experience

There is always tension and tradeoff between security and user experience. Using our cookie scheme with SSL session IDs embedded, if a user closes their web browser and later opens their web browser to access the same web site, the cookie will be invalid due to the change of SSL session IDs and the user needs to be authenticated with their credentials again. In fact currently for many secure web sites, such as Chase.com, if a user closes their web browser and even immediately later opens their web browser to access the same web site, the user needs to be authenticated with their credentials again. Also, using our cookie scheme with SSL session IDs embedded,

Cookie Verification
Input : A cookie
Output: TRUE if the cookie is valid; FALSE otherwise

1. Compare the cookie's expiration time and the server's current time. If the cookie has expired, then return FALSE.
2. Compute the encryption key as follows:
   k=HMAC(user name|expiration time, sk)
3. Decrypt the encrypted data using k.
4. Compute HMAC(user name|expiration time|data|session ID, k), and compare it with the keyed-hash message authentication code of the cookie. If they match, then return TRUE; otherwise return FALSE.

Fig. 3. Cookie verification.

when the server expires the SSL session, the user needs to be authenticated with their credentials again because the cookie becomes invalid. Nevertheless, for web applications that such user experience is undesirable, we can exclude the SSL session ID from our cookie scheme, although this sacrifices security to a certain degree.

3. Wordpress usage

Our secure cookie protocol has been used by Wordpress since version 2.4 [12]. Wordpress is a widely used open source content management system [15]. It is the largest self-hosted blogging tool in the world and it has been used on millions of sites and seen by tens of millions of people every day. Since version 2.4, Wordpress adopted our cookie scheme as follows:

Wordpress made two modifications to our secure cookie scheme. First, Wordpress eliminates data in their cookie scheme. This is because they only use cookies for authentication. They do not use cookies to track states. Second, Wordpress did not incorporate an SSL session ID (or key) in computing the HMAC. This is because Wordpress installations mostly run on non-SSL hosts [13].

4. Implementation and performance evaluation

In this section, we discuss the implementation and performance evaluation of our secure cookie scheme.

4.1. Implementation details

To evaluate the performance of our secure cookie scheme, we implemented the following five schemes in PHP:

1. *The insecure cookie scheme*: In this scheme, no cookie has any message authentication code or has any encryption, i.e., each cookie contains only the following three fields in plain text: user name, expiration time, and data. The purpose of implementing this is to provide a baseline for performance comparison.
2. *Fu's cookie scheme with low-level confidentiality*: The scheme has been seen in Fig. 1.
3. *Our cookie scheme with low-level confidentiality*: This scheme is basically the one in Fig. 2 except that the field $(\text{data})_k$ is replaced by plain text (data).
4. *Fu's cookie scheme with high-level confidentiality*: Since Fu's cookie scheme does not provide high-level confidentiality, for performance evaluation purposes, we assume that the data field of a cookie is encrypted by the server key, although this is not a good idea in preventing the server key from being cracked, as we have discussed previously.
5. *Our cookie scheme with high-level confidentiality*: This scheme has been seen in Fig. 2.

Here we discuss some details of the implementation of the above cookie schemes. We use HMAC-SHA1 (in the Crypt_HMAC package of the PEAR PHP library) as the keyed-hash message authentication code function. We use the Rijndael-256 algorithm (in the mcrypt library) through ECB (Electronic Code Book) mode as the encryption algorithm. The server key consists of 256 bits.

4.2. Performance evaluation

The goal of our performance evaluation is to compare the performance of our cookie scheme with other cookie schemes on both the client and server sides. Our test environment consists of a medium-loaded commercial web server, which uses a 2.4 GHz Celeron processor, 512 MB RAM, and runs Microsoft Windows 2003 Standard Edition, IIS 6.0, PHP 4.3.10 and MySQL 3.23; and a client, which uses a 2.8 GHz Pentium 4, 512 MB RAM, and runs Red Hat 3.0. The server and the client are connected by a dedicated Gigabit link with a 0.9 ms round-trip time.

We run each of the five cookie schemes over SSL connections between the client and the server. For each scheme, the client makes 10,000 successive requests to the server, where each request has an attached cookie. We measure the performance on both the client side and the server side.

4.3. Client side performance

On the client side, we measure the average latency from when a request (with a cookie) is sent out to when a response (with a new cookie) is received. In our implementations, the server creates a new cookie after it receives a valid cookie. In other words, the client side latency that we measured consists of:

1. the time for *transferring* a request with a cookie from the client to the server, and
2. the time for the server to *verify* the cookie in the request, and
3. the time for the server to *create* a new cookie, and
4. the time for *transferring* a response with a new cookie from the server to the client.

The overall size of each request including HTTP headers is on average 1 KB. The experimental results for the client side performance are shown in Fig. 5.

From the data in Fig. 5, we can see that the performance of our secure cookie scheme is very close to that of Fu's cookie scheme, while our secure cookie scheme provides much better security.

4.4. Server side performance

On the server side, we measure the average processing time for verifying a cookie (that was sent along a request from the client to the server) and creating a new cookie (that will be sent along a response from the server to the client).

Similarly, the overall size of each request including HTTP headers is on average 1 KB. The experimental results for the server side performance are shown in Fig. 5.

Similarly, from the data in Fig. 6, we can see that the performance of our secure cookie scheme is very close to that of Fu's cookie scheme.

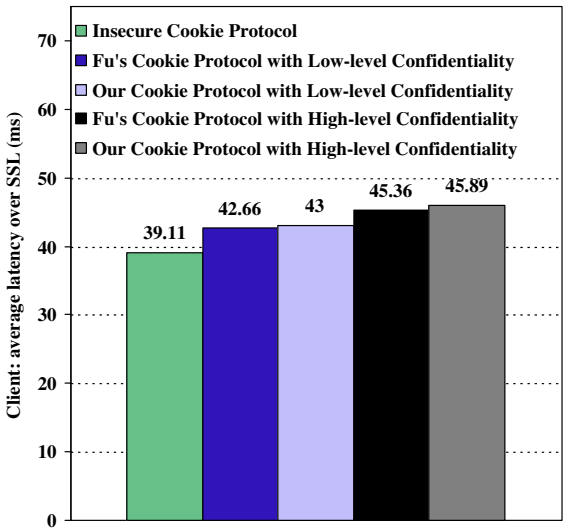

Fig. 5. Client side performance comparison.

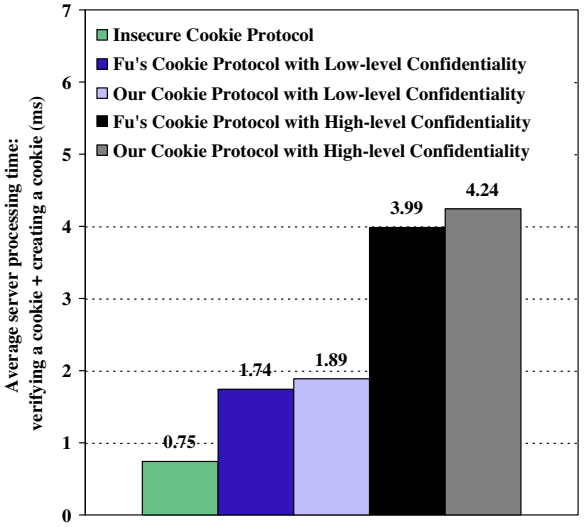

Fig. 6. Server side performance comparison.

5. Related work

In this section, we examine previous secure cookie schemes and compare them with our secure cookie scheme.

Fu et al. investigated several home-brew cookie schemes and demonstrated their vulnerabilities, leading to Fu's more secure cookie scheme [5]. As discussed in Section 2, Fu's cookie scheme has three major limitations. First, it does not have a mechanism for providing high-level confidentiality. Second, it is vulnerable to cookie replay attacks. Third, it does not provide mechanisms for key updating.

Park and Sandhu proposed a cookie scheme that uses a set of inter-dependent cookies such as a name cookie, a life cookie, a password cookie, a seal cookie, etc. [10]. As discussed in Section 2.2, the mechanism for providing confidentiality in this scheme is inefficient. Next, we examine the following three authentication mechanisms proposed by Park and Sandhu [10]: address-based authentication, password-based authentication, and digital-signature-based authentication. Using address-based authentication, each cookie set has an IP cookie that contains the IP address of the client. A server authenticates a received cookie set by verifying whether the cookie set is from the IP address in the IP cookie. This authentication mechanism has three problems. First, it is vulnerable to IP spoofing. Second, a client's IP address may be dynamically changing. Third, a client may use a NAT (Network Address Translator) or a proxy server and therefore may share the same (global) IP address with other clients in the same domain. Using the password-based authentication, each cookie set has a password cookie that contains the message digest of the client's password. A server authenticates a received cookie set by verifying whether the value of the password cookie is correct, which requires database lookups. Using the digital-signature-based authentication, each time a client wants to make an HTTP request to a server, the client first generates a signature cookie that contains a time stamp and the client's signature of the time stamp. Secondly, the client sends the HTTP request together with the cookie set issued by the server to the client and the signature cookie created by the client. The server authenticates a received cookie set by verifying the signature cookie. This authentication mechanism is difficult to deploy because it assumes every client has a public key and a private key. Moreover, this authentication mechanism is expensive because it requires both database lookups (for a client's public key) and public key cryptography.

Xu et al. presented a cookie scheme that is used by a server to store the credit card information of every client [16]. As discussed in Section 2.2, this scheme could not correctly verify multiple simultaneous requests with the same cookie. In addition, this scheme is inefficient because of the overhead of database lookups and updates for verifying each cookie.

Blundo et al. proposed a web authentication scheme that uses encrypted cookies [3]. A downside of this cookie scheme is that it requires a server to do database lookups in verifying every received cookie.

We have presented a preliminary version of our secure cookie scheme [8]. However, Wordpress usage and the experimental results on the server side was not presented in [8].

Akhawe et al. recently proposed a formal model of web security based on an abstraction of the web platform and use this model to analyze the security of several sample web mechanisms and applications including those that use cookies [1].

Tappenden and Miller proposed an EBNF grammatical definition and a three-tiered testing strategy for web cookies [14].

Yue et al. proposed a system that can automatically validate the usefulness of cookies from a Web site and set the cookie usage permission on behalf of users [17]. This system helps users achieve the maximumbenefit brought by cookies, while minimizing the possible privacy and security risks.

6. Conclusions

Our contributions in this paper are threefold. First, we discover that the state-of-the-art cookie scheme by Fu et al. has the following three major problems: it does not have a mechanism for providing high-level confidentiality, it is vulnerable to cookie replay attacks, and it does not provide convenient mechanisms for key updating. Second, we present a solution to each of these problems and we present a new secure cookie scheme. Third, we conduct performance evaluation of our secure cookie scheme. The experiments show that our scheme and Fu's scheme are very close in terms of efficiency.

Acknowledgement

The authors thank the editor Dr. Marco Gruteser and the anonymous reviewers for their constructive comments and useful suggestions on improving the presentation of this work. The authors also would like to thank Jeremy Brotherton and Kiyoshi Shikuma for suggesting to replace the SSL session key by the SSL session ID in our cookie scheme.

References

[1] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, D. Song, Towards a formal foundation of web security, in: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, 2010.
[2] M. Bellare, R. Canetti, H. Krawczyk, Keying hash functions for message authentication, in: Proceedings of CRYPTO'96, LNCS, vol. 1109, 1996, pp. 1–15.
[3] C. Blundo, S. Cimato, R.D. Prisco, A lightweight approach to authenticated web caching, in: Proceedings of IEEE 2005 International Symposium on Applications and the Internet (SAINT 2005), 2005, pp. 157–163.
[4] D. Eastlake, P. Jones, Us secure hash Algorithm 1 (sha1), RFC 3174 (2001).
[5] K. Fu, E. Sit, K. Smith, and N. Feamster, Dos and don'ts of client authentication on the web. In Proceedings of the 10th USENIX Security Symposium, August 2001.
[6] H. Krawczyk, M. Bellare, and R. Canetti, Hmac: Keyed-hashing for message authentication, RFC 2104 (1997).
[7] D. Kristol and L. Montulli, Http state management mechanism, RFC 2965 (2000).
[8] A.X. Liu, J.M. Kovacs, C.-T. Huang, and M.G. Gouda, A secure cookie protocol, in: Proceedings of the 14th IEEE International Conference on Computer Communications and Networks, pp. 333–338, October 2005.
[9] Microsoft ASP.NET Authentication, http://msdn.microsoft.com/en-us/library/aa291347(v=vs.71).aspx. 2004.
[10] J.S. Park and R.S. Sandhu, Secure cookies on the web, IEEE Internet Computing 4(4) (2000) 36–44.
[11] R. Rivest, The md5 message-digest algorithm, RFC 1321 (1992).
[12] Secure Cookies and Passwords, http://ryan.boren.me/2007/12/17/secure-cookies-and-passwords/. 2011.
[13] Secure/bullet proof cookies, http://www.arthurkoziel.com/2008/05/17/securebullet-proof-cookies/. 2011.
[14] A. Tappenden and J. Miller, A three-tiered testing strategy for cookies, in: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, 2008.
[15] Wordpress, http://wordpress.org/. 2011.
[16] D. Xu, C. Lu, and A.D. Santos, Protecting web usage of credit cards using one-time pad cookie encryption, in: Proceedings of the 18th Annual Computer Security Applications Conference, pp. 51–58, December 2002.
[17] C. Yue, M. Xie, and H. Wang, Automatic cookie usage setting with cookiepicker, in: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007.

Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the IEEE and IFIP William C. Carter Award in 2004 and an NSF CAREER award in 2009. He received the MSU College of Engineering Withrow Distinguished Scholar Award in 2011. His research interests focus on networking, security, and dependable systems.

Jason M. Kovacs received his B.S. degree in computer science from the University of Texas at Austin in 2005. He is a professional web developer with many years of experience on creating robust back-end web applications, databases and user interfaces for businesses and organizations. He is also experienced in systems architecture within multiple industries, including manufacturing, insurance, education, media and social networking.

Mohamed G. Gouda obtained his Ph.D. in Computer Science from the University of Waterloo. He worked for the Honeywell Corporate Technology Center in Minneapolis from 1977 to 1980. In 1980, he joined the University of Texas at Austin where he currently holds the Mike A. Myers Centennial Professorship in Computer Sciences. He was the founding Editor-in-Chief of the Springer-Verlag journal Distributed Computing 1985–1989. His research areas are distributed and concurrent computing and network schemes.