A Secure Cookie Protocol

Alex X. Liu[1], Jason M. Kovacs
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-0233, U.S.A.
{alex, jmkovacs}@cs.utexas.edu

Chin-Tser Huang
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, South Carolina 29208, U.S.A.
huangct@cse.sc.edu

Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-0233, U.S.A.
gouda@cs.utexas.edu

Abstract

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

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 protocol. A secure cookie protocol 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 protocol should allow the server to verify that the client has been authenticated within a certain time period. Moreover, any client should not 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*.

   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 he establishes an SSL (Secure Sockets Layer) connection with the server. 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.

   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.

1 Alex X. Liu is the corresponding author of this paper.

0-7803-9428-3/05/$20.00 ©2005 IEEE.

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 protocol may achieve: low-level confidentiality and high-level confidentiality.
   (a) *Low-level Confidentiality*: A secure cookie protocol 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 protocol 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 protocol runs on top of SSL.
   (b) *High-level Confidentiality*: A secure cookie protocol 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 [8]. 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 protocol should be able to support either low-level confidentiality or high-level confidentiality configurations.
3. **Integrity**: A secure cookie protocol 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 protocol 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 protocol, besides the above security requirements, we also need to consider the issues of efficiency and deployability. As for efficiency concerns, a secure cookie protocol 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 protocol should avoid requiring a client to possess a public key and a private key, which is currently impractical to assume.

Several cookie protocols have been proposed [2, 5, 8, 10]; however, none of these protocols are completely satisfactory. The cookie protocol in [5] has three weaknesses: it does not have a mechanism for providing high-level confidentiality, it is vulnerable to cookie replay attacks, and its mechanism for defending against volume attacks is inefficient and non-scalable. The three authentication mechanisms of the cookie protocol in [8] are either ineffective or difficult to deploy. The cookie protocols in [2] and [10] are inefficient because they require database lookups in verifying a cookie.

In this paper, we propose a secure cookie protocol that is effective, efficient, and easy to deploy. In terms of effectiveness, our secure cookie protocol provides all of the above four security services. In terms of efficiency, our secure cookie protocol does not involve any database lookup or public key cryptography. In terms of deployability, our secure cookie protocol can be easily deployed on an existing web server, 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 protocol in detail. In Section 3, we discuss the implementation of our secure cookie protocol and its performance. In Section 4, we review and examine existing cookie protocols. We give concluding remarks in Section 5.

2. Secure Cookie Protocol

The state of the art of secure cookie protocols is the one presented by Fu et al. in [5]. In this section, we first examine this protocol, which we refer as *Fu's cookie protocol*. We show that this protocol has three major problems, and we give a solution to each of them. Finally, we present our secure cookie protocol. The notations used in this section are listed in the following table.

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

Table 1: Notations

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 cheap 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 [1, 4, 6, 9].

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

2.1. Fu's Cookie Protocol

Fu's cookie protocol is shown in Figure 1.

user name|expiration time|data
|HMAC(user name|expiration time|data, sk)

Figure 1. Fu's Cookie Protocol

In this cookie protocol, 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, which is expressed as seconds past 1970 GMT, indicates when this cookie will expire and the server should reject it from a client.
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) 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.

Fu's cookie protocol has three major security weaknesses. First, it does not have a mechanism for providing high-level confidentiality. Second, it is vulnerable to cookie replay attacks. Third, its mechanism for defending against volume attacks is inefficient and non-scalable. Next, we detail these three weaknesses and give an efficient and secure solution to each of them.

2.2. Problem 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 protocol should encrypt the data field of each cookie. Now the question is this: which key should a server use for this encryption?

Fu's protocol does not provide an answer to this question. There is only one key involved in Fu's protocol, 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, but, it is prudent to avoid any such possibility.

The cookie protocol proposed in [10] 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 protocol proposed in [8] 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. The downside of this solution is the verification of every cookie involves public key cryptography, which makes the cookie protocol 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. Problem II: Replay Attacks

Fu's cookie protocol 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 his hard disk, an attacker may steal it using Trojans, worms, or viruses. An attacker may steal a cookie by launching a Denning-Sacco Attack [3]. In such an attack, an attacker first collects a large number of messages that are encrypted by the same SSL session key, and then obtains the SSL session key using various methods. 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 key into the keyed-hash message authentication code of a cookie, i.e., to use HMAC(user name|expiration name|data|session key, sk) as the keyed-hash message authentication code of each cookie. Therefore, a cookie becomes session specific. Even if an attacker steals a cookie, he cannot successfully replay it since the session key is known only to a legitimate client and the server that creates the cookie.

2.4. Problem III: Volume Attacks

Using Fu's cookie protocol, a server uses the same server key in computing the HMAC of every cookie. This makes the protocol potentially vulnerable to volume attacks. To launch volume attacks, an attacker first collects a large number of HMACs of different messages where every HMAC is computed using the same key; then the attacker discovers the key using various methods. So far, HMAC-MD5 and HMAC-SHA1 are not believed to be vulnerable to volume attacks [6], however, as stated in [5], it is prudent to defend against such attacks.

The solution proposed in [5] for defending against volume attacks is to periodically change the server key. However, this solution is inefficient and non-scalable. First, 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. Second, to make sure that the current server key is changed to a new random key that has not been used before, the server program has to be manually updated and recompiled during key exchanges. Clearly, this is not a scalable solution if the number of server programs to be updated is large.

Our solution to this problem is efficient and scalable. 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.

2.5. Our Secure Cookie Protocol

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

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

Figure 2. Our Secure Cookie Protocol

Note that all of the four fields of user name, expiration time, $(\text{data})_k$, and HMAC(user name|expiration time|data|session key, 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 to use them to compute HMAC(user name|expiration time, sk). Note that the field HMAC(user name|expiration time|data|session key, k) is used by our protocol to provide authentication, integrity, and anti-replay.

Our secure cookie protocol can be configured to provide either high-level or low-level confidentiality. The one shown in Figure 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 protocol is shown in Figure 3. Note that if FALSE is returned, the cookie is deemed invalid and the client must login in again using his user name and password.

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 key, k), and compare it with the keyed-hash message authentication code of the cookie. If they match, then return TRUE;
   otherwise return FALSE.

Figure 3. Cookie Verification

3. Implementation and Performance

In this section, we discuss the implementation and performance issues of our secure cookie protocol. To measure the performance of our secure cookie protocol in comparison to Fu's cookie protocol, we developed the following four implementations in PHP: (1) Fu's cookie protocol with low-level confidentiality, (2) Our cookie protocol with low-level confidentiality, (3) Fu's cookie protocol with high-level confidentiality, (4) Our cookie protocol with high-level confidentiality.

Since Fu's cookie protocol does not provide high-level confidentiality, for performance evaluation purposes, we assume that the data field of a cookie is encrypted with the server key. In order to provide a baseline for comparison, we also implemented an insecure cookie protocol in which 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.

Here we discuss some details of the implementation of the above cookie protocols. 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 160 bits. Every keyed-hash message authentication code output consists of 320 bits.

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 protocols over SSL connections between the client and the server. For each protocol, the client made 10,000 successive requests to the server, where each request has an attached cookie. We measure on the client side the average time from when a request was sent to when a response was received. In our implementations, the server creates a new cookie after it receives a valid cookie. In other words, the end-to-end latency that we measured consists of: (1) the time for transferring a request with a cookie from the client to the server, (2) the time for the server to verify the cookie in the request, (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 1KB. The experimental results are shown in Figure 4.

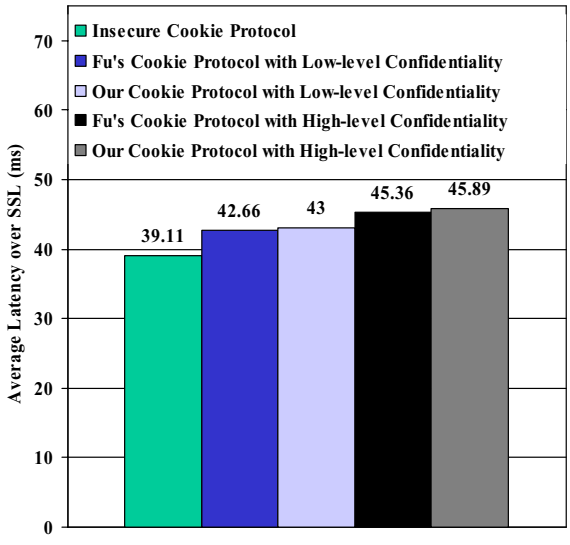

Figure 4. End-to-end Performance Comparison

From the data in Figure 4, we can see that the performance of our secure cookie protocol is very close to that of Fu's cookie scheme, while our secure cookie protocol provides better security.

4. Related Work

In this section, we examine previous secure cookie protocols and compare them with our secure cookie protocol.

In [5], Fu et al. discussed several home-brew cookie protocols and demonstrated their vulnerabilities, leading to Fu's more secure cookie protocol. As discussed in Section 2, Fu's cookie protocol has three major security problems: it does not have a mechanism for providing high-level confidentiality, it is vulnerable to cookie replay attacks, and its mechanism for defending against volume attacks is inefficient and non-scalable.

In [8], Park and Sandhu proposed a cookie protocol that uses a set of inter-dependent cookies such as a name cookie, a life cookie, a password cookie, a seal cookie, etc. As discussed in Section 2.2, the mechanism for providing confidentiality in this protocol is inefficient. Next, we examine the following three authentication mechanisms proposed in [8]: 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.

In [10], Xu et al. presented a cookie protocol that is used by a server to store the credit card information of every client. As discussed in Section 2.2, this protocol could not correctly verify multiple simultaneous requests with the same cookie. In addition, this protocol is inefficient because of the overhead of database lookups and updates for verifying each cookie.

In [2], Blundo et al. discussed a web authentication protocol that uses encrypted cookies. A downside of this cookie protocol is that it requires a server to do database lookups in verifying every received cookie.

5. Conclusions

Our contributions in this paper are threefold. First, we discover that the state-of-the-art cookie protocol 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 its mechanism for defending against volume attacks is inefficient and non-scalable. Second, we present a solution to each of these problems and we present a new secure cookie protocol. Third, we make a thorough performance evaluation of our secure cookie protocol. The experiments show that our protocol and Fu's protocol are very close in terms of efficiency, while our protocol provides much better security.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proceedings of CRYPTO'96, LNCS, vol. 1109,*, pages 1–15, 1996.

[2] C. Blundo, S. Cimato, and R. D. Prisco. A lightweight approach to authenticated web caching. In *Proceedings of IEEE 2005 International Symposium on Applications and the Internet (SAINT 2005)*, pages 157–163.

[3] D. E. Denning and G. M. Sacco. Timestamps in key distribution systems. *Communications of the ACM*, 24(8):533–536, 1981.

[4] D. Eastlake and 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] J. S. Park and R. S. Sandhu. Secure cookies on the web. *IEEE Internet Computing*, 4(4):36–44, 2000.

[9] R. Rivest. The md5 message-digest algorithm. *RFC 1321*, 1992.

[10] 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*, pages 51–58, December 2002.