International Journal of Computer Engineering and Information Technology

VOL. 9, NO. 12, December 2017, 308–312
Available online at: www.ijceit.org
E-ISSN 2412-8856 (Online)

Cookie-less Browsing

Mohamed Al-Ibrahim[1], Naser Al-Ibrahim[2] and Ahmad Bennakhi[3]

[1] Computing Department, KILAW, Kuwait

[2, 3] Computer Engineering Department, Kuwait University , Kuwait

[1]alibrahim@kilaw.edu.kw, [2]01naser14@gmail.com, [3]abenakhi@kisr.edu.kw

ABSTRACT

Cookies are used to help users browse the web using the stateless protocol of HTTP. These cookies can perform a variety of functions such as user authentication and history tracking. Even though these cookies are useful, malicious users may try to exploit these cookies by attempting to sniff or hijack a user's cookie. To avoid these security threats, a secure method of using cookie must be developed. This method must provide the following services: authentication, confidentiality, integrity, and anti-reply. Several methods have been proposed in the literature; however none of them seem completely acceptable. In this paper, we propose two new schemes to securely browse the web without the use of cookies while still maintaining the advantages that a cookie provides.

Keywords: *Cookies, Session-Hijacking, Web, Client-Server.*

1. INTRODUCTION

The Internet browsing revolutionized the way people access information. Since the 1990's, users attempt to access information through the web using the HTTP protocol. The Hyper-Text Transfer Protocol (HTTP) is a protocol used by a client to access a file or invoke a method located in the server. The protocol functions using request and response messages. The client initiates a request to the server. The server processes the request and then responds with resources such as HTML files and other content. Each request-response message from the server and client are independent of each other, meaning there is no relationship between a request now and a request in the future. This feature in a protocol is known as stateless. However many applications require dependency among requests instead of requests being independent. This gave rise to the implementation of cookies. Cookies are plain text files sent by the website and are saved in the users computer. These cookie files are really small where they do not exceed 4000 bytes and are typically less than 200 bytes. Cookies help an application to maintain a state. The state of a client is recorded in the cookie. Servers create and attach cookies whenever they need to remember some information from a client. The client upon receiving a cookie saves it and attaches it in every subsequent request.

Cookies can come in many forms, such as temporary session files that are deleted when the browser is closed or persistent files that are saved in client browser and are set to expire in the future at a certain date. Since the HTTP protocol is stateless, these cookies can be used to personalize a site, authenticating a user to a website by requiring the username and password, record purchases during online shopping, analyze web traffic and many other functions.

Cookies are useful in helping users browse the web but attackers exploited these cookies, making them do harm more than good. These attackers can attempt certain types of attacks such as SSL Heart-Bleed Attack to hijack a session or Man-in- the-Browser Attack to steal personal information. Some of these attackers' goals are to track other users by deploying tracing cookies. Others deploy trojans to keep sessions active even after one attempt a logout.

In a secure cookie scheme [6], the following services must be provided to ensure the security: authentication, confidentiality, integrity, and anti-reply. Authentication and confidentiality can be provided if the server can verify that a cookie belongs to a certain user within a time frame and no other user can forge this cookie, and ensure that the user and the server are the only parties that are able to read the cookie. Integrity is provided if the server is able to detect if a cookie was modified, while the anti-reply service can be provided if the server detects a previously sent cookie.

1.1 Problem Statement

In this paper, we propose two new secure scheme to browse without the use cookies: the first is statefull while the second is stateless. The problems that the schemes

trying to solve are to identify users during browsing and maintaining user's state without leaking information, while providing all the security requirements for the scheme to be considered secure. The rest of this paper is organized as follows: Section 2 browse related work in the literature. Section 3 demonstrate the secure cookieless schemes. Finally, Section 4 contains evaluation and comparison of the proposed schemes along with other approaches in the literature.

2. RELATED WORK

Different solutions were suggested by researchers; among them is the idea of a one-time cookie, it was suggested in the paper "One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens" [3]. Their idea was to replace cookies with a tokenization system that stores their cookie outside the browser's components. Their design is similar to the concept of TKIP (Temporal Key Integral Protocol) and Kerberos tickets, but instead, it's used here with tokens. Their key idea was to use different tokens each time data was exchanged, so that no token can be used more than once. \cite{dacosta2012one} is one of the best protocols, since it's immune to all sorts of SSL attacks, cross-site scripting and tracing attacks, weak token generation. The system has been tested and proven that it's faster than the conventional cookie system. Amazing as it is, the protocol has some limitations. To avoid hijacking, HTTPS has to be first used to exchange user and session credentials, otherwise data would be sent in open, exposing it to all sorts of attacks. Another limitation is that all of the credentials are stored in the browser. Malware controlling the browser could use this property to have total control over the session.

A less radical measure has been suggested by "An automatic HTTP cookie management system" [7]. Their solution involved more monitoring and diagnosing rather than starting over with a new session system, which is a smart approach if executed in a seamless manner. This paper has the potential to become a ground foundation to build any automated cookie management software.

The "Cookieless Monster" [1] paper has a wealth of data on non-cookie based fingerprinting techniques including: popular plugins, vendor-specific, font-detection, and detection of HTTP proxies. It also demonstrated that the browser vendors can offer full privacy if they wished to do so.

Cookie usage on an untrusted terminal have been discussed in a couple of papers. Among them is the "Session Juggler: Secure Web Login from an Untrusted Terminal Using Session Hijacking" [2] Sessions would be started on mobile phones or tablets then transferred to the untrusted device using QR code recognition. The application aims to solve the problems of improper log out instances and the leaking of HTTP plaintext keys.

"Eradicating Bearer Tokens for Session Management" [4] proposed an application "SecSess" that is trying to solve the exact same problem that we are trying to solve. Their solution involves the usage of the Diffie-Hellman key distribution protocol for session management. A proof of concept was also implemented using the Node.js as a framework. Instead of coming up of a solution from scratch, SecSess came up with a public key solution to stop cookies from being hijacked midsession. It's simple and they proved that it's faster than regular cookies too. The problem with it is that a man-in-the-middle attack can still take place if the attacker started data tracing from the beginning. The paper knew of this weakness and suggested that this problem can be solved by the use of TLS, but this only opens up the possibility of the various kinds of attacks that can be performed using SSL weaknesses.

3. COOKIELESS SCHEME

A cookie's identity remains unchanged most of the time, this opens up the possibility of someone else using the cookie for his own deeds, thus the term "cookie/session hijacking" is not an uncommon word to hear in the computer security community. Sites as famous as Amazon don't use https encryption unless the user logs in to the site. In this case there won't be a major security risk in case of a session hijacking attack, but it could undermine the privacy of browsing without logging in.

The emergence of numerous attacks, such as Oddjob [5] that keep a cookie session alive even though it seems for the client that it had logged out, could pose a major threat to all, especially to the banking and finance industry. Third-Party tracking cookies are terrifyingly common these days, but the average Internet user isn't even aware of them.

Not to mention cross-site scripting, where the attacker can trick the client into sending his/her cookies to the attacker's server. This type of cookie theft attack is executable since the cookies are stored in the client's browser/computer, in which our approach takes in mind and will be discussed later on in Section 4. Attackers can also exploit cross-site scripting in older browser versions by using proxy requests to gain a victim's cookie. Similarly cross-site request forgery can exploit the cookie stored to gain additional credentials to pose as another person.

The property that defines the cookie itself as stored identification data that a browser can access opens up many vulnerabilities that can be abused and misused. Cookies can also be inefficient because each browser has its own storage area for cookies. If a client decides to use

another browser to enter the same site, he won't be able to be identified.
We propose two scheme to eliminates the use of cookies and avoid the vulnerabilities that can be exploited with cookies. The first scheme allows the server to keep track of all the users using a fingerprint. The second scheme uses cryptography to allow the server to keeps track of it users without knowing who they are. Further details are provided in the following section.

3.1 Scheme **A**

To ensure security and provide authentication, confidentiality, integrity, and anti-reply our first scheme operates in the following manner. A user will request a page from a server. If its a new user, the server will create a new entry in the sessions' table containing a salt[1]. along with the client's IP address. The salt is then sent to the client and the client will combine his fingerprint[2] with the salt and hashes the result. The resulting hash is sent back to the server. Upon receiving the reply, the server will save the resulted hash to the sessions' table thus creating a session for this client.

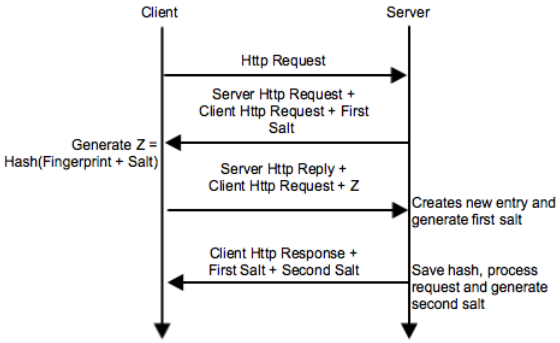

Fig. 1. Session Initiation

Fig. 1. discusses the details of the session initiation process between the client and the server. Once a session is created, any new communication between the client and the server is maintained by having the client determine the resulted hash of it's fingerprint and the salt. To avoid hijacking, the server will send two salts to the client. The first salt changes every time a new session is created. The second salt changes with every new request. The client uses the first salt to compute the first hash which is the hashed sum of the fingerprint and the first salt. Using this result with the second salt to compute the required hash for the server. Fig. 2 shows the details of a clients maintaining a session with the server.

[1] *nonce*
[2] *information collected about a remote computing device for the purpose of identification when cookies are not used*

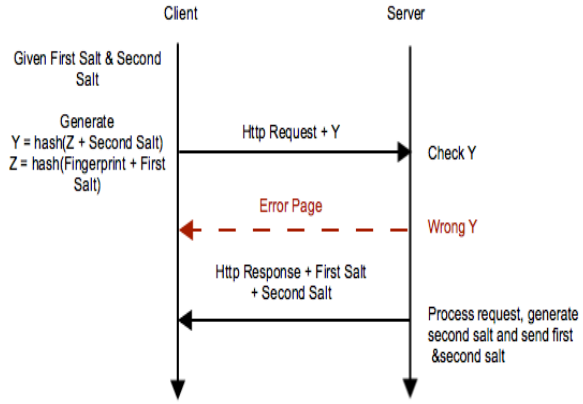

Fig. 2. Session Maintenance

3.2 Scheme **B**

To avoid maintaining server side state, this scheme uses the concept of cryptography to establish a communication between the server and the client. The scheme operates by having the server encrypting and decrypting a black box that is appended with every users request. Initially a new user requesting access from the server will not have this black box therefore the server creates this box and appends it with the reply. Any new request from the user will require him to send this black box back to the server to check the identity of this client. To generate this black box and maintain it, the server has to create a decision graph for the application its hosting. A decision graph for a web application is a graph where all the nodes in the graph are pages in the application and all the edges are links that can be accessed from that page. Once this graph is created, it is not changed unless the application is updated. Upon the creation of this decision graph, the server can append the following information in the black box to identify this user. The previous state/node where the user was, the variables created in the previous state, a timestamp, and the users fingerprint. Once the server modifies the black box to contain this new information it encrypts it with a private symmetric key known only to it and reply back to the client. A client will not be able to modify the black box since he doesn't have a key. Upon receiving a request from an existing client with a black box the server will decrypt the box and use the information inside to reply to the client. Fig. ~\ref{fig:sb} shows the details of maintaining the black box.

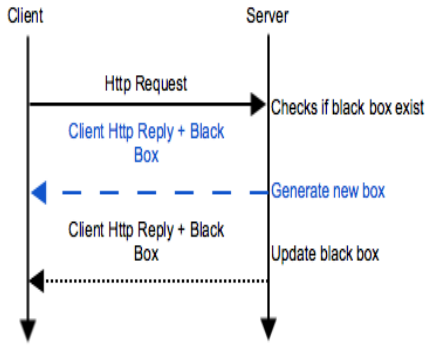

Fig. 3. Black Box Maintenance

3.3 Implementation

The implementation of both Scheme's A & B where executed using a PHP server. *Algorithm 1* and *Algorithm 2* below are the pseud-code for the scripts.

Algorithm 1: Scheme A Pseud-Code

```
1: procedure SCHEME
2: if hash is attached then
3: h←Compute hash[hash(Fingerprint+1st-Salt)+2nd-Salt]
4:     if hash = h then
5:       Generate new 2nd-salt
6:       Process user's request
7:       Reply to user
8:     end if
9:     if hash != h then
10:      Reply with error
11:    end if
12: end if
13:  if hash is not attached then
14:    Generate1st salt
15:    Request from client hash(Fingerprint+1st-Salt)
16:    Receive hash and create session
17:    Generate 2nd salt
18:    Process user's request
19:    Reply to user
20: end if
```

Algorithm 2: Scheme B Pseud-Code

```
1: procedure SCHEME
2:       if Box is attached then
3:           Decrypt box
4:           if Box content is correct then
5:             Append new information to box
6:             Encrypt box
7:             Process user's request and reply to user
8:           end if
9:       if Box content is not correct then
10:          Reply with error
11:      end if
12:    end if
13:    if Box is not attached then
14:      Create box
15:      Append information to box
16:      Encrypt box
17:      Process user's request and reply to user
18:    end if
```

In the server, a script was written to be attached to every page requested. Fig. 4 illustrate the script details.

id	ip address	first salt	hash	second salt	variables
10	192.168.1.2	1240130	637303f2ca69340022fecd9d0052948db5e4ad070eacab1a93...	149239	

Fig. 4. Server Script

4. EVALUATION

4.1 Design

Scheme A is supposed to keep track of users by storing their information in the server while not saving anything on the client side; therefore eliminating any danger of stored tracking cookies as such. To keep this scheme from saving anything on the client side, the first salt has to be sent with each request reply from the server. The first three steps of the session initiation has to be performed using HTTPS since an attacker can high jack the session if he gets Z, which is Hash(Fingerprint + 1st Salt). If an attacker listens to a communication between a client and a server midway, it won't be of any use since the latter messages will have different encryption due to the fact that the second salt is changing after each request.
Scheme B is a protocol that keeps state without saving it on either side i.e. stateless. This makes memory overflows in DoS attacks not possible. The design of this networking protocol can be used not only for sessions, but also for networking too.

4.2 Security Analysis

In Scheme A, integrity is sustained as long as HTTPS is used to initiate the session, no real breach of integrity could be taken. We also have to taken in account that there may be a possibility that two users with different fingerprints may have equal hashes based on the salt. For example, User-1(Fingerprint + 1st-Salt) = User-2(Fingerprint + 1st-Salt). This error is avoided by having the server check during the session initiation setup that no user with the same hash value does already exists. Confidentiality can be maintained by having users encrypt their messages with the Hash(Z+2nd-Salt) during the session management setup, where Z=Hash(Fingerprint+ 1st-salt). Availability is maintained by having the server reachable at any time to generate a random number and check. On the other hand, while DoS and DDoS attacks could affect the server, it's mainly out of this protocol's scope and should be prevented using other monitoring applications and hardware such as firewalls. Anti-reply of messages is prevented by using the second salt. This create a new random number to ensure freshness of the message.
In Scheme B integrity can be provided using the concept of asymmetric key cryptography. By having the clients sign the black box with their private key and having the server verifying it with their public key provides integrity. Using the decision tree provides confidentiality for both the client and server since no one other than the server could update the blackbox and the state of the blackbox is not saved but updated and then sent back. No memory overflows are possible from session registration because nothing is saved, therefore allowing more sessions to be processed in parallel and providing availability. Since the blackbox contains timestamps anti-reply of previous blackbox is not possible.

4.3 Comparison

While One-Time cookie [3] has a great session management system, the session credentials are still stored on the client's PC, allowing the session to be tracked by others. None of our schemes store any data on the client side for this purpose mainly. Therefore, what makes both of these implementations special is that session tracking is not possible. Yet, the SecSess application [4] is just as the One-Time cookie proposal stores session entrees on both the server and the client sides, which could cause memory overflows from DoS attacks on both sides while our implementations either reduce this to one side or none at all.

5. CONCLUSION

While cookies can provide numerous useful functions, they're greatly misused and are being outdated. New alternatives are being vied to replace cookies and we're suggesting two schemes that could very well take the place of cookies in keeping up sessions.

By designing a scheme that doesn't save any cookies on the client side, many malicious attacks such as cross-site scripting, memory overflows, session hijacking, and various others can be avoided. Scheme A provided the same exact functionality as cookies, except that it is way more secure. The use of temporary salts provide a shield against attackers that intercept session IDs since it would be useless to use an expired session ID and if the attacker intercepted the 2nd salt then it would be useless because he doesn't know the rest of the session ID values.
Scheme B requires no non-volatile memory whatsoever, not on the client side or the server side. It uses cryptography to deter any one from exploiting them. Hence maintaining integrity and confidentiality alive by design. The main purpose of scheme B is to maintain state while not being saved on either side.

REFERENCES

[1] F. Beyer. Cookieless monster: Exploring the ecosystem of web based device fingerprinting. 2013.
[2] E. Bursztein, C. Soman, D. Boneh, and J. C. Mitchell. Session juggler: secure web login from an untrusted terminal using session hijacking. In Proceedings of the 21st international conference on World Wide Web, pages 321–330. ACM, 2012.
[3] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. Onetime cookies: Preventing session hijacking attacks with stateless authentication tokens. ACM Transactions on Internet Technology (TOIT), 12(1):1, 2012.
[4] P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Eradicating bearer tokens for session management. In W3C/IAB workshop on strengthening the internet against pervasive monitoring (STRINT), 2014.
[5] S. Gold. The cookie monster. Computer Fraud & Security, 2011(9):12–15, 2011.
[6] A. X. Liu, J. M. Kovacs, and M. G. Gouda. A secure cookie scheme. Computer Networks, 56(6):1723–1730, 2012.
[7] C. Yue, M. Xie, and H. Wang. An automatic http cookie management system. Computer Networks, 54(13):2182–2198, 2010.