Hardened Stateless Session Cookies

Steven J. Murdoch

University of Cambridge, Computer Laboratory
http://www.cl.cam.ac.uk/users/sjm217/

Abstract. Stateless session cookies allow web applications to alter their behaviour based on user preferences and access rights, without maintaining server-side state for each session. This is desirable because it reduces the impact of denial of service attacks and eases database replication issues in load-balanced environments. The security of existing session cookie proposals depends on the server protecting the secrecy of a symmetric MAC key, which for engineering reasons is usually stored in a database, and thus at risk of accidental leakage or disclosure via application vulnerabilities. In this paper we show that by including a salted iterated hash of the user password in the database, and its pre-image in a session cookie, an attacker with read access to the server is unable to spoof an authenticated session. Even with knowledge of the server's MAC key the attacker needs a user's password, which is not stored on the server, to create a valid cookie. By extending an existing session cookie scheme, we maintain all the previous security guarantees, but also preserve security under partial compromise.

1 Introduction

Many websites require users to authenticate themselves before permitting access. Reasons include customising the appearance to meet user preferences, restricting access to confidential information, limiting who can change site configuration, and tracking who contributes to the site. The protocol used for webpage access, HTTP [1], does not provide a session layer. If needed, websites must implement a mechanism for securely linking a series of requests together and back to a user account. This paper discusses the construction of a session management system which is robust against disclosure of the authentication database.

1.1 Web Authentication

Users almost universally authenticate to websites by providing a username and password. Both are sent as a HTTP form submission, optionally encrypted with TLS [2,3]. The website will then retrieve the specified user's account details, typically from a SQL [4] database, and check if the password is correct.

It is prudent engineering practice not to record the cleartext password in the database; instead the result of a one-way function should be stored. This allows the site to verify whether a presented password is correct, but an attacker who can read the authentication database cannot directly retrieve the password. It would still be possible for such an attacker to test all common passwords, and time-space trade-offs such as rainbow tables [5] can perform this attack almost instantaneously. To resist these attacks, a per-account random value, the *salt*, should be additionally fed into the one-way function, and stored. Adding a salt makes pre-computing a dictionary infeasible (hence each account must be brute-forced individually), and hides whether two users share a password [6].

1.2 Session Management

Following successful authentication, the website must be able to link HTTP requests to the relevant user account. This is achieved by the website returning a byte string to the client on login, a *session cookie* [7]. The client will include this cookie in the headers of subsequent HTTP requests to the same domain, until the cookie is deleted by the user or it expires. Cookies may be set as persistent, and otherwise will be be deleted when the web browser exits. The structure of the cookie is opaque to the client.

There are a number of standard approaches for constructing session cookies. One common technique, supported by web frameworks such as PHP, stores session state on the webserver, in a file or database. Here, the cookie contains a randomly generated session identifier which allows the website to retrieve or save the relevant session state. The cookie contents should be a cryptographically secure pseudorandom number and be sufficiently long to make guessing a valid session identifier infeasible.

Many users will have multiple sessions associated with their account, for example one for work and one from their home PC. Also, unless users explicitly log out, the server state associated with a session must be retained until the session times out. This means that the space required on the server for maintaining state can grow much faster than the number of users. This also introduces a denial of service vulnerability, in that attackers could create many sessions and fill up the filesystem. CAPTCHAs [8] can restrict automated account creation, but it would significantly harm usability to require them for login.

It is thus desirable to implement stateless session cookies. Here, the server does not need to store any session state – all necessary information is stored in the cookie held by the client. With this approach, load balancing is easier, as session state does not need to be replicated over multiple front-end servers. This paper will discuss how to implement such cookies securely, so that attackers cannot spoof authenticated users or alter critical session data. While previous work has assumed that an attacker has no control of the server state, here we show how a limited version of the security guarantees can be retained even when the attacker is able to read the account database.

There are a number of ways in which unauthorised read access to a database can be gained. For example, a simple Google search can find several database backups for blogs unintentionally left on the web[1]. Also, one of the most common

[1] I have contacted the site operators in question, recommending they remove the backups and change their passwords.

security vulnerabilities in web applications is the SQL injection attack, which we will show can often grant read, but not write, access.

1.3 SQL Injection

Storing website data in a relational databases is a very common design choice made by developers. This approach permits convenient and efficient querying of data, and allows the now stateless front-end webserver to be replicated for scalability and redundancy. However, the use of a database introduces the risk of SQL injection attacks, one of the most problematic classes of web application vulnerabilities. Here, improperly sanitised user provided information is used in constructing a SQL query, allowing the attacker to inject malicious SQL code.

For example, suppose the following SQL query is executed by a web application, in order to retrieve the account details for a user[2].

```
SELECT * FROM wp_users WHERE user_login = '$user_login'
```

The value of `$user_login` may be chosen by the attacker, and if it contains a `'` character, the string literal will be terminated and the remainder of the value interpreted as SQL. To exploit this vulnerability the following value for `$user_login` may be chosen:

```
' UNION ALL SELECT 1,2,user_pass,4,5,6,7,8,9,10 FROM wp_users
WHERE ID=1/*
```

Now, the SQL executed will be:

```
SELECT * FROM wp_users WHERE user_login = '' UNION ALL SELECT
1,2,user_pass,4,5,6,7,8,9,10 FROM wp_users WHERE ID=1/*'
```

The result of this query will be the concatenation (union) of rows returned from the two subqueries. As no rows fulfil the `user_login = ''` expression, the only row retrieved will be where the user's ID is 1, which is by convention the site administrator. The `/*` starts a comment, preventing the trailing `'` from causing a syntax error.

As will be shown in Section 2, with Wordpress, knowing the contents of the `user_pass` column is sufficient to impersonate the respective user. If the result of the above query can be retrieved, the attacker can then exploit further security weaknesses to eventually escalate privileges to user-level, and potentially root access, on the web server.

At this point, it is useful to note the structure of the queries above – the attacker can only append SQL into the existing string. In particular, the attacker cannot convert a SELECT query (for reading data) into UPDATE or INSERT (which modify data). The SQL syntax does support executing multiple queries

[2] This example is a simplified version of the exploit code for CVE-2007-2821, a SQL injection vulnerability in Wordpress 2.1.3, written by Janek Vind: `http://www.milw0rm.com/exploits/3960`

in a single call, by separating them with a `;` character, however the standard database API rejects such queries[3]. Effectively, the attacker has read-only access to the database.

In many cases, the ability to read the database is sufficient to totally compromise the security of a web application. Web servers are, for security reasons, commonly prevented from writing any data to the filesystem, other than via the database engine. Thus all secrets, whether randomly generated or entered into the management web interface, are at risk from SQL injection vulnerabilities.

As with buffer overflows, the theory of preventing SQL vulnerabilities is well understood but poorly applied in practice, especially in legacy applications. Identifying potentially dangerous code is difficult, especially when user input is processed by several layers of escaping and unescaping, by partially undocumented libraries, before the string is passed to the database engine. Prepared statements [9] improve matters, but these cannot be used when the user input defines the structure of the query itself.

It appears that SQL injection vulnerabilities are inevitable in large web applications – even if they do not exist when the system is written they could easily be introduced later by less-experienced programmers or carelessly written libraries. Therefore, following the principle of defence in depth, in addition to efforts to eliminate vulnerabilities, the application should be structured to limit the harm caused by exploits.

Previous proposals for stateless session cookies fail completely if the database can be read, for example through SQL injection. This is understandable, given the common attitude that a website vulnerable to SQL injection is a hopeless case and so there is no need to consider further layers of defences. Nevertheless, we have shown some cases where an attacker has read access, but may find gaining write access more difficult, or even impossible. This paper will show how to leverage this read-only property into limiting the potential damage.

2 Weakness of Existing Proposals

Due to the lack of stateless session cookies in standard web frameworks, a wide variety of ad-hoc solutions have been developed. Fu et al. [10] showed that several of these were seriously flawed, due to weak or non-existent cryptography, using a counter when a cryptographically secure pseudorandom number is needed, and inappropriate padding. Their paper demonstrates how the schemes were reverse-engineered, without any access to the server code, simply by observing the cookies returned and sometimes through generating specially constructed usernames. It then describes how the schemes can be exploited, including spoofing cookies for chosen accounts and even extracting the system-wide key from one website.

Another ad-hoc solution is implemented by Wordpress, where the authentication cookie is of the form MD5(MD5(*password*)). The account database stores

[3] `http://www.php.net/mysql_query`

MD5(*password*) in the `user_pass` column, so on subsequent accesses, this value can be hashed a further time and compared with the cookie. This approach suffers from three main weaknesses, which are the subject of CVE-2007-6013 by the present author [11]. Firstly, no salt is applied before storing the hashed password in the database, easing brute force attacks. Secondly, if the attacker can read the hashed password in the database, it is trivial to generate a valid cookie. Thirdly, a cookie will never cease to be valid – if the database is compromised or a cookie is leaked, the affected users must change their password to a different one.

In their paper, Fu et al. [10] make a number of suggestions for improving web authentication, including proposing an improved structure for stateless authentication cookies:

$\texttt{exp=}t\texttt{\&data=}s\texttt{\&digest=}\mathrm{MAC}_k(\texttt{exp=}t\texttt{\&data=}s)$

Here, t is the expiry time for the cookie and s is the state the web application needs to be maintained, such as the username or the user capabilities. The digest is a message authentication code, such as HMAC-SHA-256, under a key known only to the web server. This scheme prevents a valid cookie being generated by an attacker and also prevents existing cookies being modified. Liu et al. [12] extended this proposal to also encrypt s and optionally bind the cookie to a particular TLS session. However, in either scheme, if the MAC key is compromised, for example through an SQL injection attack or insecure backup, an attacker may spoof cookies for any account, until the compromise is detected and the key revoked.

3 Stateless Session Cookies

The motivation for the scheme proposed in this paper is to maintain the security properties of the previous work, while also remaining secure even if the attacker has read access to the database. In essence, we suggest including an iterated hash of the user's password in the database and its pre-image in the session cookie. Thus, someone with the correct password can calculate a valid cookie, and this can be verified using the database entry, but it is infeasible to generate a cookie given access to the database alone.

In the following section, we use the following recursive definitions, based on the fallback password hashing algorithm of the `phpass` library [13]:

$$a_0(\mathit{salt}, \mathit{password}) = \mathrm{H}(\mathit{salt}||\mathit{password})$$
$$a_x(\mathit{salt}, \mathit{password}) = \mathrm{H}(a_{x-1}(\mathit{salt}, \mathit{password})||\mathit{password})$$

Where *salt* is per-account, cryptographically secure pseudorandom number, long enough to resist brute force attack (e.g. 128 bits), and *password* is the user password. $\mathrm{H}(\cdot)$ is a cryptographically secure hash function (e.g. SHA-256).

(1) Account creation: On requesting an account be created, the user specifies the desired username and password. The web site then generates the random salt, and calculates the authenticator $v = \mathrm{H}(a_n(\mathit{salt}, \mathit{password}))$ both of which it stores in the database. The public value n is the hash iteration count (e.g. 256).

(2) Login: To log in, a user presents their username and password. The web site retrieves the user account details, including the salt and authenticator. Using the supplied password and retrieved salt it calculates $c = a_n(\mathit{salt}, \mathit{password})$ and compares it to the stored authenticator v. If $\mathrm{H}(c) \neq v$ the user is denied access and c is discarded. If $\mathrm{H}(c) = v$ the web site concludes that the supplied password was correct and returns a cookie, constructed as per the Fu et al. scheme, but with an extra field `auth`:

`exp=`t`&data=`s`&auth=`c`&digest=`$\mathrm{MAC}_k($`exp=`t`&data=`s`&auth=`$c)$

(3) Subsequent accesses: Following login, as the client requests further pages, the server reads the submitted cookie, checks the MAC, extracts c, and compares $\mathrm{H}(c)$ with the authenticator v from the database. If they match, access is granted.

3.1 Security

In the conventional threat model, an attacker does not have access to the authentication database, so does not know the salt, password or MAC key and here our proposal performs just as well as the Fu et al. scheme. Even though it is likely that many users will select poor passwords, the inclusion of a large salt prevents operation (3) being of help in brute-forceing the password. This property is desirable because rate-limiting, to resist online brute-force attacks, need only be applied to the login procedure (2), not every page that requires authentication. This could be particularly valuable in load-balanced situations with one login server and multiple application servers – only the login server need retain state on the number of failed login attempts.

If a cookie is retrieved, for example from a compromised or stolen computer, or through XSS [14], the attacker may use it to log in, until the cookie expires. The MAC prevents the expiry time from being modified. However, as the attacker does not know the salt, the cookie cannot be used to confirm whether a particular brute-force guess at the password is correct. It is for this reason that the salt in the scheme must be fairly large. In conventional password hashing schemes the salt is only to make a precomputed dictionary infeasible to store, and so a few tens of bits are adequate. In this scheme the salt is additionally used to prevent the attacker learning any information about the password from a cookie, so must be as large as a cryptographic key, e.g. 128 bits.

If the attacker is able to read the database, e.g. through unsecured backups or SQL injection, it can discover a user's authenticator, salt and global MAC key. This is still insufficient to generate a valid cookie, as doing so would be equivalent to discovering the pre-image of a hash output. Of course, the attacker can still brute force the password, and if it is weak the attacker can gain access. This task is made more difficult by the salt (preventing a rainbow-table attack) and is further slowed down by iterating the hash function n times (e.g. 256).

With knowledge of the MAC key, an attacker can alter cookies, so user capabilities cannot be safely stored there and instead must reside in the account database. If an attacker compromises the MAC key and can intercept a cookie for a privileged user, its expiry date can be extended until the compromise is detected and key revoked. Regularly rotating the MAC key would require the attacker to gain access to a recent copy, limiting the vulnerable period.

An attacker with write-access to the database is outside the threat model of this scheme. With such access, the attacker could simply create a new account with chosen password or escalate an existing account to high privilege. We see no way to defend against such an adversary in the general case.

3.2 Efficiency

The scheme proposed in this paper is not significantly more computationally expensive than one based around the Fu et al. [10] proposal. At account creation and login, the only additional step is one invocation of $\mathrm{H}(\cdot)$, as the iterated computation is present in any salted password authentication scheme. If the salt needs to be extended, there will be a increase in size of data hashed, but as the salt is only added once, this increase is negligible. On each subsequent access, where authentication is required, an extra invocation of $\mathrm{H}(\cdot)$ is needed, in addition to the two already required for HMAC verification.

We also require a database read to retrieve the authenticator and user access rights. In contrast, the Fu et al. scheme can operate independently from the database after the login stage, because the site-wide MAC key can be cached in memory. Our scheme introduces more complexity in load-balanced environments because front-end servers need access to the database, however, unlike standard session cookies, only read access is required so maintaining consistency is easier and on-disk indices need not be updated.

4 Variations

A large variety of possibilities for variations on this scheme are possible, and may be desirable in certain deployment environments. Any sufficiently secure password hashing mechanism may replace $a_n(\mathit{salt}, \mathit{password})$, provided the salt size is adequate to also prevent offline brute-force password-recovery attacks on cookies. For example `bcrypt` [15], with its 128-bit salt, would be a good candidate. If possible, the password hashing function should be written in an efficient compiled language, allowing a higher value of n, so as to reduce the advantage of attacker with a custom implementation.

If the available password hashing functions do not permit a sufficiently long salt to be added, an alternative is to encrypt the cookie under a server key before sending it to the client [12]. This means that the cookie alone is still not helpful for attempting a brute-force password recovery attack. Both the extended-salt and encrypted authenticator schemes are vulnerable to brute force, if the user's salt or encryption key respectively are available to the attacker.

We have not selected the encrypted authenticator option because standard web libraries do not come with symmetric encryption libraries, possibly due to export regulations. Also, if the salt is sufficiently short for there to be collisions, a user who has the same password on multiple sites might have the same authenticator on some of them. In this case, if an attacker compromises one site, obtaining a user's cookie along with the cookie encryption key, he can replay the cookie to a different site for which he has compromised the MAC key if the user has both the same password and salt. This weakness can be defended against by selecting a long enough salt such that collisions are unlikely, or by incorporating the site URL in the calculation of the authenticator (for example, by appending it to the salt when calculating v).

Fu et al. suggest a number of other hints for maintaining secure web sites, which complement the authentication proposal in this paper. Their recommendations include prohibiting weak passwords, requiring re-authentication before changing passwords, using HTTPS to encrypt login, and binding cookies to IP addresses. We add a further recommendation, of storing two MAC sub-keys if possible – one in a file and one in the database, which are combined at runtime. To compromise the key an attacker would need to obtain both sub-keys which mitigates certain vulnerabilities.

In addition to security, usability is another important consideration for an authentication scheme. Enforcing an expiry time on cookies reduces the risk of cookie compromise, but requires users to re-authenticate periodically. This could be especially annoying to users if they compose a long blog post, only to find out that their cookie has expired when they submit the form, potentially losing their changes. A simple way to reduce this risk is to require users re-authenticate a few hours before the expiry time if a GET request is made, but to permit POST requests up to the hard deadline.

5 Conclusion

In this paper, we have described how an attack resulting in read-only access to a website authentication database is a plausible scenario, motivated by the examples of a badly-protected database backup and SQL injection vulnerabilities. We have shown how to harden existing stateless session cookie schemes, which were previously vulnerable to such attackers, thereby greatly limiting the potential damage. The new scheme we have presented has negligible overhead, when compared to existing proposals, and can leverage existing password authentication libraries, yet provides good security guarantees even when the attacker has read access to the full website state.

Acknowledgements

Thanks are due to Richard Clayton, Markus Kuhn, the attendees at the Security Protocols Workshop (especially Mark Lomas), and the contributors to Light Blue Touchpaper, for their valuable discussions on the topics discussed here.

References

1. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF (June 1999)
2. Rescorla, E.: HTTP over TLS. RFC 2818, IETF (May 2000)
3. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, IETF (April 2006)
4. JTC 1/SC 32: Information technology – database languages – SQL. ISO/IEC 9075:2006 (2003)
5. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In Boneh, D., ed.: Advances in Cryptology – Crypto 2003. Volume 2729 of LNCS., Springer (August 2003) 617–630
6. Morris, R., Thompson, K.: Password security: a case history. Communications of the ACM **22**(11) (November 1979) 594–597
7. Kristol, D., Montulli, L.: HTTP state management mechanism. RFC 2109, IETF (February 1997)
8. von Ahn, L., Blum, M., Hopper, N.J., Langford, J.: CAPTCHA: Using hard AI problems for security. In Biham, E., ed.: Advances in Cryptology – Eurocrypt 2003. Volume 2656 of LNCS., Springer (May 2003) 294–311
9. Fisk, H.: Prepared statements. MySQL Developer Zone (October 2004) `http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html`.
10. Fu, K., Sit, E., Smith, K., Feamster, N.: Dos and don'ts of client authentication on the web. In: Proceedings of the 10th USENIX Security Symposium, Washington D.C., US (August 2001)
11. Murdoch, S.J.: Wordpress cookie authentication vulnerability (November 2007) CVE-2007-6013 (candidate) `http://www.cl.cam.ac.uk/~sjm217/advisories/wordpress-cookie-auth.txt`.
12. Liu, A.X., Kovacs, J.M., Huang, C.T., Gouda, M.G.: A secure cookie protocol. In: Proceedings of the 14th IEEE International Conference on Computer Communications and Networks. (October 2005) 333–338
13. Solar Designer: Portable PHP password hashing framework (2006) `http://www.openwall.com/phpass/`.
14. CERT Coordination Center: Malicious HTML tags embedded in client web requests. Advisory CA-2000-02, CERT/CC (February 2000) `http://www.cert.org/advisories/CA-2000-02.html`.
15. Provos, N., Mazières, D.: A future-adaptable password scheme. In: USENIX Annual Technical Conference, Monterey, California, US (June 1999) 81–92