Getting web authentication right
A best-case protocol for the remaining life of passwords

Joseph Bonneau

jcb82@cl.cam.ac.uk
University of Cambridge

Abstract. We outline an end-to-end password authentication protocol for the web designed to be stateless and as secure as possible given legacy limitations of the web browser and performance constraints of commercial web servers. Our scheme is secure against very strong but passive attackers able to observe both network traffic and the server's database state. At the same time, our scheme is simple for web servers to implement and requires no changes to modern, HTML5-compliant browsers. We assume TLS is available for initial login and no other public-key cryptographic operations, but successfully defend against cookie-stealing and cookie-forging attackers and provide strong resistance to password guessing attacks.

1 Introduction

The vast majority of large websites rely on passwords for user authentication [3]. Due to the originally stateless nature of HTTP and the extremely low adoption of HTTP Digest Authentication, nearly all websites using password authentication rely on HTTP cookies to cache authentication decisions. Creating and validating session cookies securely has historically been a major source of bugs, with many websites implementing ad-hoc protocols with basic cryptographic flaws [4].

However, the precise practice of verify user passwords and managing secure sessions over the existing technology stack of HTTP/TLS with HTML and JavaScript has attracted relatively little attention from the cryptography and security research communities. Instead the focus has largely been on whole-sale replacements for the current architecture, requiring either vastly different user behavior, new client-side software, or significantly re-designed servers, and often all three. While we would agree that such fundamental changes are needed, we expect that web authentication will continue to operate using the current technology only for at least five to ten years. It is worthwhile to get authentication right in the current framework, even while the research community is actively seeking its replacement.

In this work we seek to outline a complete protocol for secure web sessions which is:

- As secure as possible using already-deployed technology
- As simple and efficient as possible given the first goal

For the first goal, we specifically rule out any solution which requires changes in user behavior or new software to be installed, including updates to current-generation browsers. Our proposal is not fundamentally new in that many elements of it are either directly analogous to elements in other proposals, or are straightforward constructions to those familiar with cryptographic literature. However, we do not believe any currently deployed web servers deploy the complete defense-in-depth we propose, including in-browser password hashing to prevent against buggy server software, storage of credentials in a server-side database to limit the damage from database compromise, cookie creation which is not possible with read-only database access, and active computation of a MAC with each web request to prevent cookie-theft.

2 Previous work

Fu et al. proposed a basic scheme in 2001 to remedy the problems they observed in a survey of common implementations, namely:

$$\text{auth\_cookie} \quad = \quad \text{time\_stamp } t || \text{user\_data } u || \mathbf{MAC}_k(t, u)$$

This basic framework has the advantages of being stateless on the server side and secure against a passive network observer. It represents a reasonable basic password implementations and most carefully-designed web servers implement a close variation of it.[1] However, this scheme is completely vulnerable to an attacker who captures the cookie, either through cross-site scripting or over an unprotected wireless network. It is also vulnerable to an attacker who gains read access to the server's databases, for example via SQL injection.

Murdoch proposed a hardened version of this scheme [10] designed to protect against an attacker with read-only database (which can occur as a result of SQL injection). This is achieved by using a pre-image of the stored password hash which can be checked against the final password hash stored in the database with each request. However, this scheme is still vulnerable to cookie theft, and allows an attacker with databases access to indefinitely extend the validity of a stolen cookie.

In a different vein, Adida proposed a scheme called SessionLock [1] to protect specifically against cookie theft by storing a session identifier in the browser's fragment identifier. The fragment identifier is never transmitted to the server, protecting against cookie-theft on an insecure network, or "sidejacking," but not protecting against cross-site scripting attacks. However, SessionLock is stateful on the server side and many important password-management details are left out of the scheme.

Finally, there are a number of more complicated proposals [7,6,9,13], all of which require either modifications to existing browsers or public-key operations performed outside the scope of the TLS protocol, both of which we consider

[1] Many minor tweaks with no security relevance, such as splitting the fields up into multiple cookies, are commonly seen in practice.

to be too heavyweight to gain widespread adoption. Additionally, few of these proposals take an integrated approach to password authentication, incorporating difficult issues such as preventing guessing attacks [11] and user-probing attacks [3].

3 Our proposal

3.1 Notation

We assume cryptographic primitives for semantically symmetric encryption, message authentication codes (MAC), and collision-resistance hashing. We denote $\mathbf{AE}_k(m)$ to be the authenticated encryption[2] of plaintext m under key k and $\mathbf{AE}_k^{-1}(m)$ to be the verified decryption of m under key k.

We assume a primitive hash function $\mathbf{H}$, and denote

$$\mathbf{H}^k(x) = \mathbf{H}(k||x)$$

allowing us to easily come up with different random functions with equivalent security to $\mathbf{H}$. We further denote $\mathbf{H}^k_n(x)$ to be an iterated hash,[3] specifically:

$$\mathbf{H}^k_1 = \mathbf{H}^k(1||x)$$
$$\mathbf{H}^k_n = \mathbf{H}^k\left(n||x||\mathbf{H}^k_{n-1}(x)\right)$$

3.2 Enrolment

Our scheme is designed so that the user never submits a password in the clear to the server, not even over an encrypted TLS session. We consider this to be prudent practice to protect against server mistakes[4] which may leak the plaintext password. We take the further unusual step of not storing usernames in the clear on the server, making it more difficult for a user compromising the database to export the list of usernames.

To enrol with username u and password p with server[5] s, a user computes via JavaScript:[6]

$$x = \mathbf{H}^{\mathrm{X}}_{\ell_1}(u||p||s)$$

[2] Authenticated encryption can be implemented as encrypt-then-MAC, or a combined mode such as EAX or GCM.

[3] The iterated hash is used purely to increase the difficulty of brute-force attacks. The purpose of folding the hashed data x and round number n in before each hash iteration is to prevent pre-computation of any long sequences of $\mathbf{H}^k$ from being useful.

[4] Hashing the password in-browser also makes insider attacks more difficult, as they require modifying the JavaScript on the login pages, instead of modifying any servers behind the SSL gateway.

[5] The inclusion of s prevents collisions between multiple sites implementing the same scheme. It could be either the domain name, or a 128-bit site-specific random number to allow the possibility of domain name changes.

[6] Our preliminary experiments indicate that iterated hasing using SHA-256 is feasible in modern browsers. We were able to hash at a rate of $\sim$ 50 kHz on desktop PCs, and

and sends x along with u to the server, with ℓ_1 a configurable security parameter.[7] The server then stores the following values in its databse:

$$y = \mathbf{H}^{\mathrm{Y}}_{\ell_2}(u||s) \qquad z = \mathbf{H}^{\mathrm{Z}}(u||x)$$

The advantage of this formulation is that usernames are not stored in the clear in the server's database, preventing a database compromise from leaking the entire list of usernames.[8] For practical reasons including keeping a count of password attempts and password recovery, it is necessary to include a hash of u, slightly limiting the privacy provided if the adversary has a large set of candidate email addresses to test. However, by using a relatively high security parameter ℓ_2 which is only computed on enrolment, incorrect log-in, or password recovery, this search can be made more difficult.[9] In ordinary use, the user's entry can be looked up by the second column, $\mathbf{H}^{\mathrm{Z}}(u||x)$ if a database index is maintained for this column as well.

On first glance, this scheme seems to violate common practice by storing $\mathbf{H}^{\mathrm{Z}}(u||x)$ which is derived from the user's password with no random salt. However, this hash incorporates both the username and the site-specific string s. As long as the site-specific identifier is unique and the site doesn't allow multiple accounts with the same username, this will mean that the hash function applied to the password p is in fact unique, eliminating the need for a salt.

3.3 Login

For normal login, the user's browser collects u and p, re-computes $x = \mathbf{H}^{\mathrm{X}}_{\ell_1}(u||p||s)$, and sends x and u to the server over a secure TLS session. The server then re-computes $z = \mathbf{H}^{\mathrm{Z}}(u||x)$ and checks for its existence in the database. With a collision-resistant hash function, it is impossible for this value to exist in more than one row, so a simple existence check is sufficient. It is not useful to submit a value of x stolen from another user, since u is included in the hash to bind the result to the correct user.

If the calculated z is found in the database, the server returns to the user a fresh random encryption key K_u and an authentication token:

$$a = \mathbf{AE}_{K_{\mathrm{S}}}(K_u, u, x, t, d)$$

where K_{S} is a secret key for the server, t is an expiration time, and d is any additional data associated with the session. Critically, we assume that the au-

$\sim$ 1 kHz on mobile devices. Since this calculation is only performed during log-in we consider a 1 s delay to be acceptable, which will allow for at least 1,000 iterations.

[7] Common practice would be to pick the hash iteration count $\ell_1 \approx 1,000$. Our scheme has the advantage that this computation is performed in-browser, the server never needs to compute iterated hashes.

[8] In practice, usernames are often email-addresses and it is important to prevent adversaries from easily determining if an email address is registered with the website.

[9] It can also be offloaded to the client, as with the iterated hash of the password.

thenticated and encrypted token a is stored as an `HTTP-only` cookie, while K_u is stored in HTML5 local storage.[10]

3.4 Site interaction

After successful login, all future requests to the site are MAC'ed with K_u. It is assumed that the majority of site interaction will not take place over TLS, making this MAC the only verification of the user's intentions. This requires a small JavaScript routine to dynamically add to all HTTP and AJAX requests a MAC parameter $\mathbf{MAC}_{K_u}(p)$ on all request parameters p, a technique originally proposed in the SessionLock protocol [1]. This MAC is sufficient to ensure the integrity of any request, but to prevent relay attacks it is also necessary to include a per-request timestamp parameter.

In addition to the request-specific MAC, the authentication token a is sent as a cookie in the normal way. In fact, because it was declared `HTTP-only`, this is the only way a can be sent, it is not accessible by the in-page JavaScript. Given a cookie a, the server computes $\mathbf{AE}_{K_\mathrm{S}}^{-1}(a)$, checks the expiration time t, and then accepts the request as authenticated. It can then check its access-control lists to verify the request using the username u, the session-data d and the request-specific parameters p.

3.5 Optimisations

The server can also optionally re-compute $z = \mathbf{H}^\mathrm{Z}(u||x)$ with the values of u, x recovered from a and verify that there is a row in the database with z. This optional check prevents specifically against an attacker who has has read-only database access and has stolen K_S. Such an attacker will be unable to forge cookies with the proper x unless they know the user's password. The fact that such a check is optional in our system means security and performance can be more finely tuned. In the Murdoch scheme [10], this check is required with every access, limiting the appeal of the protocol as a "stateless" web protocol. However, our system allows the check to be skipped with some high probability, or perhaps only checked before extremely sensitive actions are performed.

A second optimisation is to include an extended-validity authentication cookie $a^* = \mathbf{AE}_{K_\mathrm{S}}(K_u, u, x, t^*, d)$, with $t^* > t$, and mark this cookie both `HTTP-only` and `secure`. The `secure` flag means this cookie will only be be sent over TLS, making it significantly harder for an adversary to steal. Of course, since we have assumed the majority of site interaction will not take place over TLS, a^* will not be sent by default. If a user sends an authentication cookie a with an expired timestamp t, however, the server can re-direct them to a special TLS server

[10] It is also possible to "store" K_u in the fragment identifier, which was a significant part of the complexity in [1]. We assume that a sufficient super-majority of today's browsers implement HTML5 local storage that this technique is now obsolete. We have omitted the details, but this technique can still be implemented to deal with legacy browsers.

which can receive and verify a^* and then re-issue a with an extended timestamp. This process can be done with no user interaction, saving the difficulty and risk of re-typing a password. A practical set-up might be to have a^* last for several weeks, as current persistent log-in cookies frequently do, and to have a expire after ten minutes or one hour. This technique can greatly reduce the vulnerability of a sidejacked cookie, while still limiting the frequency of TLS connections to a very low number, and to a singled dedicated server.

3.6 Password recovery

We have so far ignored the difficult but important procedure of recovering user passwords when they are forgotten. The simplest procedure for achieving this is by sending a one-time reset link over email [5]. This process fits naturally with our scheme and should be sufficient for most purposes.

4 Security properties

As mentioned, our scheme makes it difficult for an attacker to retrieve usernames from the database. It also requires per-user brute-force for each password in the database, as hashed passwords always include a username in the hash. Our cookies only contain a hash of the user's password, but this is encrypted with the server key K_{S}. If an attacker has stolen this, they have likely stolen the account database as well, allowing them to brute-force values of z stored in the database.

Cookie forgery is prevented by the requirement of knowing $x = \mathbf{H}^{\mathrm{X}}_{\ell_1}(u||p||s)$ and K_{S} to compute a valid cookie. This means that an attacker must gain both read-access to the database and the user's password in order to forge a cookie.

The scheme also offers very strong protection against cookie theft, similar to SessionLock. The storage of K_u in local storage means it is never transmitted. It can be stolen using cross-site scripting, however the authentication token a is marked as an `HTTP-only` cookie, meaning it cannot be accessed via JavaScript. Thus, to steal credentials in our system an attacker must both steal a by "sidejacking" the cookie from an unprotected access point, and also steal K_u via cross-site scripting. Even if credentials are stolen in this way, the expiration date cannot be changed. This is a very high barrier to credential theft, than that enabled by previous systems that don't require changes to the browser.

References

1. Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 517–524, New York, NY, USA, 2008. ACM.
2. Carlo Blundo, Stelvio Cimato, and Roberto De Prisco. A Lightweight Approach to Authenticated Web Caching. In *Proceedings of the The 2005 Symposium on Applications and the Internet*, pages 157–163, Washington, DC, USA, 2005. IEEE Computer Society.
3. Joseph Bonneau and Sören Preibusch. The password thicket: technical and market failures in human authentication on the web. *WEIS '10: Proceedings of the Ninth Workshop on the Economics of Information Security*, June 2010.
4. Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 19–19, Berkeley, CA, USA, 2001. USENIX Association.
5. Simson L. Garfinkel. Email-Based Identification and Authentication: An Alternative to PKI? *IEEE Security and Privacy*, 1(6):20–26, 2003.
6. Mohamed G. Gouda, Alex X. Liu, Lok M. Leung, and Mohamed A. Alam. SPP: An anti-phishing single password protocol. *Computer Networks*, 51(13):3715–3726, 2007.
7. Ari Juels, Markus Jakobsson, and Sid Stamm. Active cookies for browser authentication. In *In 14th Annual Network and Distributed System Security Symposium (NDSS 07*, 2007.
8. Alex X. Liu, Jason M. Kovacs, Chin-Tser Huang, and Mohamed G. Gouda. A secure cookie protocol. In *14th International Conference on Computer Communications and Networks*, 2005.
9. Chris Masone, Kwang-Hyun Baek, and Sean Smith. WSKE: web server key enabled cookies. In *Proceedings of the 11th International Conference on Financial cryptography and 1st International conference on Usable Security*, FC'07/USEC'07, pages 294–306, Berlin, Heidelberg, 2007. Springer-Verlag.
10. Steven J. Murdoch. Hardened Stateless Session Cookies. In *Sixteenth International Workshop on Security Protocols* , 2008.
11. Benny Pinkas and Tomas Sander. Securing passwords against dictionary attacks. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 161–170, New York, NY, USA, 2002. ACM.
12. Guy Pujolle, Ahmed Serhrouchni, and Ines Ayadi. Secure session management with cookies. In *Proceedings of the 7th international conference on Information, communications and signal processing*, ICICS'09, pages 689–694, Piscataway, NJ, USA, 2009. IEEE Press.
13. Tim van der Horst. pwdArmor: Protecting Conventional Password-Based Authentications. In *Annual Computer Security Applications Conference*, 2008.