Dr Cookie and Mr Token - Web Session Implementations and How to Live with Them *

Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi

Università Ca' Foscari Venezia
name.surname@unive.it

Abstract

The implementation of web sessions is a somewhat anarchic and largely unstructured process. Our goal with the present paper is to provide a disciplined perspective of which are the relative strengths and weaknesses of the most common techniques to implement web sessions, with a particular focus on their security. We clarify common misconceptions in the recent "cookies vs tokens" debate and we propose a more useful classification of web session implementations, based on where session information and session credentials are stored. We then propose a new implementation technique for web sessions which combines the strengths of existing web technologies to overcome their weaknesses and we successfully deploy our solution on top of WordPress and the Auth0 library for web authentication to demonstrate its feasibility.

1 Introduction

Both HTTP and its encrypted variant HTTPS are *stateless* protocols, which means that different HTTP(S) requests normally look unrelated to the web server receiving them. Most of the web applications, however, need to keep track of state information across multiple HTTP(S) requests, e.g., to provide authenticated access to their private area or to preserve payment information along an e-commerce transaction. Web applications thus routinely build *sessions* on top of HTTP(S) by requiring browsers to present enough information to bind together multiple requests, e.g., under the same user identity.

Unfortunately, the implementation of web sessions is a somewhat anarchic and largely unstructured process. In their session implementations, web developers are free to choose which information browsers should present when interacting with their web applications, as well as how this information is used at the server side to implement the session abstraction. The most common implementation technique binds session information to unpredictable session identifiers in a server-side database and stores the session identifiers in web browsers using *cookies*. When a cookie including a session identifier is attached to a HTTP(S) request, the web application can use it to lookup the database and restore the appropriate session information. Though this simple solution flawlessly works for many web applications, web session implementations in the wild are extremely complex and variegated for a number of reasons, like the need to integrate together multiple web development frameworks, and it may be complicated to understand their inner workings [5]. A more recent trend in the implementation of web sessions, which is growing in popularity, makes use of *JSON Web Tokens* (JWTs). JWTs are cryptographically signed (and possibly encrypted) JSON-based credentials which assert a number of unforgeable *claims*. For example, a web application could generate a token with the claim "logged in as administrator", as well as a number of other claims for storing state information, and deliver it to the browser of an authenticated user. This way, session information is stored at the browser side,

*This research is supported by the MIUR project ADAPT.

rather than at the server side, which voids the need of a centralized database and simplifies the deployment of the web application over multiple independent servers.

As it usually happens when implementation freedom is rampant, it did not take too much time before the "cookies vs tokens" debate started spreading over the web. There is a wealth of blog posts and discussions in online communities like Stack Overflow which argue for the use of one of the approaches over the other one, but they are often biased, suffer from a lack of rigor or provide only a very partial point of view. Our goal with the present paper is to provide a more disciplined perspective of which are the relative strengths and weaknesses of the most common techniques to implement web sessions, with a particular focus on their security.

1.1 Contributions

In this paper, we make the following contributions: (*i*) we clarify common misconceptions in the recent "cookies vs tokens" debate and we propose a more useful classification of web session implementations, based on where session information and session credentials are stored, which we use to perform a critical assessment of the relative strengths and weaknesses of the different implementation techniques; (*ii*) we propose a new implementation technique for web sessions which combines the strengths of existing web technologies to overcome their weaknesses; and (*iii*) we successfully deploy our solution on top of WordPress and the Auth0 library for web authentication to demonstrate its feasibility.

2 Web Sessions: Core Concepts

2.1 Storing Session Information

The first degree of freedom in web session implementations is related to the storage of *session information*, like the user identity, the content of the shopping cart or the preferred language of choice for the website. This information can be stored either at the server or at the browser.

In a *stateful* session, the session information is stored at the server side and the browser only stores a "pointer" to this information. For example, the server S may generate a unique session identifier Xa33YkW1, which is used to index a database containing session information and supplied to the browser B. When B sends back the session identifier, S uses it to perform a query to its database. The result of the query includes the required session information, like the identity of the user and her preferred language; this information is used to dynamically assemble a personalized web page, which is returned in response to the browser.

In a *stateless* session, the session information is stored at the browser side, so the server does not need to rely on a database for session management. To prevent tampering at the browser side, the server uses cryptography to protect the integrity (and possibly the confidentiality) of the stored session information. For example, the server S may use the signing key $sk(K)$ to produce a signature of the identity of the user and her preferred language. When the browser B provides back the signature, S uses the verification key $vk(K)$ to check its integrity and restore the session like in the previous case.

2.2 Storing Session Credentials

The second degree of freedom in web session implementations is related to the storage of *session credentials* at the browser side. By "session credential" we mean everything which unambigu-

ously identifies a web session and can be used to start the process of resuming it, e.g., the session identifiers typical of stateful sessions or the signatures used to implement stateless sessions.

2.2.1 Cookies

The most common technology to store session credentials are *cookies*. Roughly, a cookie is a key-value pair generated by a web application and sent to the user's browser to be stored therein; the browser then automatically attaches the cookie to the next HTTP(S) requests sent to the web application, which can inspect the cookie value to start resuming the session.

Cookies have a well-defined *scope*, since they are attached by default to all the HTTP(S) requests directed to the domain and the sub-paths of the HTTP(S) endpoint setting them. For instance, a cookie set by `https://www.example.com/` will be attached to any HTTP(S) request to `example.com`. Path visibility can be configured by means of the Path attribute, but most notably cookies can be shared among multiple domains by means of the Domain attribute. Specifically, a page can set cookies for a parent domain as long as the latter does not occur in a list of public suffixes[1]: these cookies are shared between the parent domain and all its sub-domains. For instance, a page at `www.example.com` can set a cookie with the Domain attribute set to `.example.com`, which is sent with all requests to sub-domains like `mail.example.com` and `accounts.example.com`.

2.2.2 Web Storage

A more modern alternative to cookies is *web storage*, a novel feature of HTML5. Web storage is a browser-side storage mechanism which can be accessed by means of JavaScript APIs. Compared to cookies, it offers larger storage capacity and it is only intended for browser-side scripting, i.e., web storage data is not automatically attached to every HTTP(S) request and just stays in the browser, though it is possible to implement different behaviors in JavaScript if desired.

Web storage offers two storage areas: *local* storage and *session* storage. The only difference is that data stored in the local storage persist until explicitly deleted, while the content of the session storage is only made available to a given window or tab (depending on the browser); once the window or tab is closed, the storage is deleted. We will not distinguish between local storage and session storage in the rest of the paper.

2.2.3 Same-Origin Policy

The *same-origin policy* (SOP) is the baseline defense mechanism of web browsers and it mediates accesses to both cookies and the web storage. It prevents the theft of sensitive credentials by scripts of untrusted websites, e.g., credentials owned by `bank.com` cannot be accessed by `evil.com`. More precisely, an *origin* is defined as a triple including a scheme, a hostname and a port, like (`https`, `example.com`, `443`). A script can read or write the contents of the web storage of a given origin only if it runs exactly in the same origin. The same idea applies to cookies, but with the important difference that cookies implement a more relaxed notion of origin, which does not provide isolation by scheme and port [1]. For example, a script running in the origin (`http`, `example.com`, `80`) is allowed to read cookies set by an HTTPS response from (`https`, `example.com`, `443`). Further relaxations can be implemented by means of the Domain attribute, as explained above.

[1] The common domain suffix must not occur in a list of public suffixes at `https://publicsuffix.org/`.

2.3 Threat Model for Web Sessions

Web security traditionally considers different threat models, which we reuse in the present paper. The *web attacker* runs a set of malicious websites hosting arbitrary attacker-controlled contents and can exploit content injection vulnerabilities on trusted websites, i.e., he may be able to place maliciously crafted contents in otherwise benign web pages. A more powerful variant of the web attacker, known as the *related-domain attacker*, can host his malicious websites on a sibling domain of the target website [3]. As we anticipated, websites hosted on sibling domains can set cookies which are visible to each other and indistinguishable from other cookies which are set so that this sharing across domains is not desired. Finally, the *network attacker* is able to block, inspect and corrupt all the HTTP traffic exchanged on the network. He can also block outgoing HTTPS connections, but he cannot read or modify the HTTPS traffic, because we assume the adoption of trusted certificates.

3 How to Implement Web Sessions?

3.1 Misconceptions

A common attempt in technical blogs and web communities is comparing cookies and tokens as two different mechanisms to implement web sessions. However, such a comparison is ill-founded, because, irrespective of which technology is chosen, the lack of native support for sessions on HTTP(S) encourages web developers and authors of web development frameworks to come up with custom techniques for session management. In practice, both cookies and tokens can be used to implement sessions in many different ways. A much better way to classify and reason about web session implementations is just based on where session information and session credentials are stored, as we do after clarifying two common misconceptions.

3.1.1 Cookies = Stateful and Tokens = Stateless

Traditionally, cookies are associated to stateful sessions, while tokens are associated to stateless sessions, but actually the need of storing and accessing session information at the server side is orthogonal to the use of cookies and tokens. Since cookies are just a generic storage mechanism, stateless sessions can be easily implemented on top of them. Indeed, we found real services using cookies to implement stateless sessions, with WordPress being a notable example.

Dually, stateful sessions may be built on top of tokens, most notably to implement *token revocation*. Although tokens typically include an expiration date, there is no way to invalidate them before expiration without implementing a stateful blacklisting mechanism at the server side. Revoking tokens may be useful for several reasons, e.g., when a password theft is reported, when a user logs out or when a user's password is changed. We found plugins to implement token revocation for popular token-based web development frameworks, like Express[2].

3.1.2 Sharing across Domains

A common misconception is that cookie-based sessions work well with single domains and their sub-domains, while tokens are easier to share between domains. The reason underlying this observation is that cookies have a well-defined scope based on domain matching, while tokens do not have a fixed scope and are shared via the implementation of custom JavaScript logic.

[2]The plugin is available at https://github.com/auth0/express-jwt

Still, although cookies have well-defined scoping rules respected by web browsers, they can still be accessed by JavaScript and shared across domains just like tokens.

Ironically, we observe that tokens can be actually harder to share than cookies when they are stored in the web storage, because read and write accesses to the web storage are subject to stricter access control checks than the cookie jar - cookies are not isolated by scheme and port. For instance, if a token is saved in the web storage by a script at `http://example.com`, it cannot be read by scripts from `https://example.com`, while cookies set by `http://example.com` are visible to `https://example.com` and vice-versa.

3.2 Stateful Sessions vs Stateless Sessions

The main arguments in favor of stateless sessions are efficiency and scalability. In stateful sessions, every authenticated request requires a database access to be processed, which is costly and makes server-side replication and load balancing more complex to deal with. On the flip side, stateless sessions typically require to exchange more data upon communication, because session information is entirely stored at the browser side. From a security perspective, however, there is no strong argument to prefer stateful sessions over stateless sessions or vice-versa, because robust programming practices are known for both.

When stateful sessions are implemented, web developers should ensure that session information is only bound to unpredictable session identifiers. This is important to guarantee that valid session identifiers cannot be forged or detected by brute-forcing [7]. Popular web development languages like PHP and ASP provide native support for stateful sessions based on robust routines for the generation of session identifiers. Moreover, session identifiers should be refreshed upon successful password-based authentication to prevent session fixation attacks [9].

When stateless sessions are preferred, web developers should rely on appropriate signature and encryption algorithms to ensure the integrity and, possibly, the confidentiality of the session information [7, 11]. Moreover, they should adopt best practices to protect their cryptographic keys and rely on hardening techniques against key compromise [12].

3.3 Cookies vs Web Storage

3.3.1 Programming Convenience

Cookies are significantly easier to use than web storage when it comes to storing session credentials, because they have a well-defined semantics for web browsers based on their scoping rules. When credentials are stored inside a cookie, web sessions can be implemented entirely at the server side, because browsers will automatically perform the task of attaching the cookie. If instead credentials are stored in the web storage, the web application must implement enough JavaScript logic to attach them to HTTP(S) requests.

This also limits the attachment of the credentials to requests which can be programmatically generated by JavaScript, i.e., using the XMLHttpRequest API. If a user clicks an external link pointing to the web application, for instance among the results of a search engine, no JavaScript logic will be triggered by the browser and no credentials will be attached to the request. Hence, any ongoing session will be broken when clicking the link, unless a script on the landing page forces a page refresh to provide the credentials.

3.3.2 Confidentiality of Credentials

Both the cookie jar and the web storage are protected by the same-origin policy, which mediates both read and write accesses, but does not constrain cross-origin communication. This means

that, normally, any script running in the same origin which set the session credentials is entitled to read them and communicate them to any HTTP(S) endpoint on the web. Session credentials are then vulnerable to exfiltration via script injection (XSS) or accidental leaks over HTTP, which allow the hijacking of legitimate user sessions (impersonation attacks).

Although the cookie jar relies on a more relaxed notion of origin than the web storage, the former provides better confidentiality guarantees than the latter, because cookies can be protected using the HttpOnly and Secure attributes [1]. Cookies marked as HttpOnly are not visible to scripts, which prevents their exfiltration via XSS. Cookies marked as Secure are only exposed to scripts running in HTTPS pages and never attached to HTTP requests, hence they are never leaked to attackers sniffing the HTTP traffic. The combined use of the two attributes provides strong confidentiality guarantees to cookies, even against network attackers [4].

3.3.3 Integrity of Credentials

It is well-known that cookies do not provide strong integrity guarantees against related-domain attackers and network attackers, given their more relaxed notion of origin [1, 3]. An attacker owning `evil.example.com` can set cookies with the Domain attribute set to `.example.com`, which would be sent to `bank.example.com` and would be indistinguishable from other cookies directly set by `bank.example.com`. A network attacker can easily force the browser to access `http://bank.example.com` and set cookies to be sent to `https://bank.example.com`, even if `bank.example.com` was only served over HTTPS. These attacks can be exploited to force the user's browser in the attacker's session, which is a subtle form of session hijacking.

Cookie integrity can be enforced by placing web contents so that security-sensitive cookies cannot be set by untrusted sibling domains and by using HSTS [8] to block HTTP connections to the domain and sibling domains. Unfortunately, this is likely too complex for most web developers and the adoption of HSTS still lags behind [10]; as a matter of fact, even major websites do not yet properly protect the integrity of their cookies, with subtle security implications [14].

3.3.4 Integrity of Requests

Since cookies are attached by default to every HTTP(S) request, they enable cross-site request forgery attacks (CSRF) [2]. In these attacks, a web page at `evil.com` sends a state-changing request to `bank.com`, e.g., to authorize a payment of 1,000$ to the attacker's account. If the user's browser has an ongoing session with `bank.com`, it will automatically attach to the request the cookies containing the user's session credentials, thus authorizing the payment on the user's behalf. There are many techniques to defend against CSRF attacks, though none of them is optimal and most of them are not straightforward to implement correctly [6].

If the session credentials are stored in the web storage, they are not attached by the browser to any request by default; rather, the web developers are completely in charge of implementing which credentials should be attached to which requests. This means that CSRF attacks are automatically prevented: if authenticated cross-site requests are really needed, they should be explicitly supported by the implementation of appropriate JavaScript logic.

4 Reconciling Cookies and Web Storage

We have discussed the relative strengths and weaknesses of cookies and web storage for saving session credentials. We discuss here two possible ways to combine cookies and web storage to implement more robust web sessions.

4.1 Proposed Solutions

Our key idea is to take advantage of the confidentiality guarantees offered by cookies, without sacrificing the integrity assurances and the protection against CSRF granted by the use of web storage. Specifically, we propose two ways to combine these existing web technologies.

In our first scheme, we store the session credentials in the web storage and an HMAC of the session credentials inside a cookie. The HMAC is computed using a symmetric key which never leaves the server. The cookie is protected with the HttpOnly attribute and, if full HTTPS support is available, also with the Secure attribute. When a request including valid session credentials is received, the website verifies the HMAC of the credentials before processing it. If no valid HMAC is attached to the request, the request is discarded.

In our second scheme, we store the session credentials inside a cookie and a cryptographic hash of the session credentials inside the web storage. The cookie is protected with the HttpOnly attribute and, if full HTTPS support is available, also with the Secure attribute. When a request including valid session credentials is received, the website verifies the hash of the credentials before processing it. If no valid hash is attached to the request, the request is discarded.

4.2 Security Analysis

Both schemes prevent CSRF attacks by construction, because part of the information needed to authenticate the requests is saved in the web storage and thus is not automatically attached to cross-site requests. Moreover, both schemes protect against user impersonation attempts by the attacker. In the first scheme, even if session credentials are leaked from the web storage, e.g., via XSS, their HMAC is not, because it is stored in a cookie protected with security attributes ensuring its confidentiality. Notice that the HMAC cannot be computed by the attacker, unless the symmetric key used to construct it is leaked, which would require the attacker to compromise the server. In the second scheme, the session credentials are directly stored in a cookie protected with security attributes ensuring their confidentiality and they cannot be reconstructed from the hash, because cryptographic hash functions are non-invertible.

Finally, observe that both schemes also prevent malicious attempts to force the user's browser in the attacker's session by compromising the integrity of the session credentials, unless the target website is affected by an XSS vulnerability. This is because the same-origin policy ensures the integrity of the web storage against cross-origin write attempts, thus compensating the weak integrity guarantees provided by cookies alone.

4.3 Deployment Considerations

Though both schemes support the same security guarantees, there are a number of deployment considerations which should be taken into account when comparing them. The first important observation to make is that there are several cases where authenticated cross-site requests should not to be considered CSRF attempts and are actually desired, e.g., users would like to stay authenticated to a website when clicking a hyperlink pointing to its homepage in the results of a search engine. The security policy enforced by the second scheme is easier to relax to support these cases, because session credentials are stored inside a cookie and thus attached by default to HTTP(S) requests; in contrast, in the first scheme only an HMAC of the session credentials is available in the cookie, which does not contain enough information to restore the session. With the second scheme, relaxing the security policy for selected web pages is straightforward, since it is enough to skip the check of the hash for those pages. This can be implemented

without changing the web application code, for instance by using a web application firewall like ModSecurity or by writing an Apache module which performs the task.

Relaxing the security policy of the first scheme is more complicated. The most natural way to achieve this is by implementing enough JavaScript logic to force the attachment of the session credentials to selected HTTP(S) requests, but this may require a significant effort and a good understanding of the web application code. A simpler solution is based on *service workers* [13], a fancy utility available in modern web browsers. A service worker is a sort of a browser-side proxy which can be registered via JavaScript to intercept requests to (and responses from) a given web origin. The intercepted requests can be modified before they exit the browser: in our case, we can use a service worker to attach the session credentials to selected HTTP(S) requests, based on the web origin which originated them - an information which is made available to the service worker inside the Referer header. Notice that the Referer header can be changed via JavaScript, but only same-origin referrers are allowed to be set[3].

Though this analysis seems to suggest that the second scheme is easier to implement than the first one, the main problem in the deployment of the second scheme is that session credentials are not available to JavaScript, because they are stored inside a cookie protected with the HttpOnly attribute. Though accessing session credentials at the browser side is uncommon for stateful sessions, it may be useful for stateless sessions, e.g., when user information is stored inside a JWT. In these cases, the first scheme is the only available solution to deploy.

5 Implementing Our Solution

We retrofitted two existing open-source applications to improve the security of their sessions based on the authentication schemes proposed in the previous section. We comment here on our deployment experience.

5.1 WordPress

WordPress is the world's most used Content Management System, powering about the 28% of the Internet websites[4]. Sessions in WordPress are built on top of cookies. The core of WordPress is protected against CSRF attacks by means of random *nonces*, a standard solution which prevents the sending of valid cross-origin requests which may produce dangerous side-effects at the server [2]. We did not check whether CSRF protection is correctly in place in the full core of WordPress, but we realized that an important part of WordPress is not protected by default: its *plugins*. WordPress plugins can leverage the protection system of the WordPress core by using APIs to thwart CSRF attempts, but their security is completely left to developers and should not be taken for granted. Moreover, the integrity of the WordPress cookies is not guaranteed, most notably because WordPress does not enforce the use of HSTS by default and the use of its "multi-site" feature may open the room to related-domain attackers.

In the case of WordPress, we chose to implement our second authentication scheme, because session credentials are already stored in cookies. We developed a WordPress plugin which extends login responses to include a fresh cookie `cookie_hash` containing a hash of the credentials and to register a new service worker. The plugin also injects a script which reads the value of `cookie_hash` and communicates it to the service worker using the postMessage API; the service worker then saves the hash value in the web storage and extends each request to the WordPress core to include it as a GET parameter. Finally, the plugin changes the WordPress core to make

[3] https://hacks.mozilla.org/2016/03/referrer-and-cache-control-apis-for-fetch/
[4] Data retrieved on 02/10/2017 from https://wordpress.org/

it check the validity of the hash before processing incoming requests. We successfully tested our plugin on a fresh WordPress 4.8.2 installation and we will proceed soon to its public release.

5.2 Auth0

Auth0 is a popular identity management platform, offering a unified access interface to a huge number of identity providers implementing single sign-on systems, like Facebook and Google. Auth0 offers its authentication services by means of a JavaScript library, which performs the single sign-on process with the chosen identity provider and stores in the web storage a JWT containing unencrypted authentication data signed with RSA-SHA256. Web developers are left in charge of choosing how to use the JWT for session management at the server side.

Since the Auth0 library stores session credentials in the web storage, we chose to implement the first authentication scheme. Specifically, we patched the Auth0 library to provide the JWT to a (customizable) server endpoint and we implemented an Apache module which performs two tasks. First, the module listens for JWTs at the designated server endpoint: when a JWT is received, the module computes its HMAC and returns it to the browser inside a fresh cookie `jwt_hmac` - future requests to the endpoint providing the same JWT will be discarded. Then, the module automatically performs the check of the HMAC on every incoming request before forwarding it to the protected web application. We successfully tested our implementation by hosting on Apache an example single page application.

This implementation significantly strengthens session security by leveraging our scheme, though it does not provide bullet-proof protection against session hijacking. An attacker who is able to steal the JWT before it is sent by the user's browser to the designated server endpoint may be able to acquire a valid HMAC for the user and intrude the session. Nevertheless, observe that the attack surface for session hijacking was significantly reduced: in the current Auth0 implementation, an XSS attack on any page at any time would allow an attacker to hijack the session. With our implementation, instead, an attacker can only hijack the session if he is able to steal the session credentials exactly when they are first established, before they are used to get a valid HMAC. This means that the attacker must perform an XSS attack at the right time and exactly on the page hosting the Auth0 library to hijack the session.

6 Conclusion

The implementation of web sessions is a largely unstructured process, which the wide availability of partial and biased information on informal web documentation does not help streamlining. With the present paper, we proposed a clear-cut classification of web session implementations to perform a critical discussion of the strengths and weaknesses of popular solutions. We then put forward a new implementation technique based on a combination of cookies and web storage, where we exploit the distinctive strengths of these web technologies to overcome their limitations. Our solution has good security properties and proved to be easy to implement on existing services like WordPress and the Auth0 library for web authentication.

The quest for more robust and easier to deploy web sessions is still ongoing and there are several avenues for future work. For instance, designing and implementing libraries for web session management which are secure by construction is an important research direction, which we are investigating in ongoing work. A particularly intriguing challenge is complementing these libraries with tools which automatically retrofit the security of existing web session implementations, thus providing improved session security to the largest possible audience.

References

[1] Adam Barth. HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265, 2011.

[2] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*, pages 75–88, 2008.

[3] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. In *Web 2.0 Security & Privacy Workshop (W2SP 2011)*, 2011.

[4] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[5] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *TWEB*, 9(3):15:1–15:30, 2015.

[6] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. Lightweight server support for browser-based CSRF protection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 273–284, 2013.

[7] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The dos and don'ts of client authentication on the web. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, 2001.

[8] Jeff Hodges, Collin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS). http://tools.ietf.org/html/rfc6797, 2012.

[9] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1531–1537, 2011.

[10] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[11] Alex X. Liu, Jason M. Kovacs, and Mohamed G. Gouda. A secure cookie scheme. *Computer Networks*, 56(6):1723–1730, 2012.

[12] Steven J. Murdoch. Hardened stateless session cookies. In *Security Protocols XVI - 16th International Workshop, Cambridge, UK, April 16-18, 2008. Revised Selected Papers*, pages 93–101, 2008.

[13] The W3C Consortium. Service workers. https://www.w3.org/TR/service-workers-1/, 2016.

[14] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies lack integrity: Real-world implications. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 707–721, 2015.