One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens

Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad and Patrick Traynor
Converging Infrastructure Security (CISEC) Laboratory
Georgia Institute of Technology
{idacosta@, schakradeo@, mustaq@cc., traynor@cc.}gatech.edu

Abstract

HTTP cookies are the de facto mechanism for session authentication in web applications. However, their inherent security weaknesses allow attacks against the integrity of web sessions. HTTPS is often recommended to protect cookies, but deploying full HTTPS support can be challenging due to performance and financial concerns, especially for highly distributed applications. Moreover, cookies can be exposed in a variety of ways even when HTTPS is enabled. In this paper, we propose *One-Time Cookies* (OTC), a more robust alternative for session authentication. OTC prevents attacks such as session hijacking by signing each user request with a session secret securely stored in the browser. Unlike other proposed solutions, OTC does not require expensive state synchronization in the web application, making it easily deployable in highly distributed systems. We implemented OTC as a plugin for the popular WordPress platform and as an extension for Firefox and Firefox for mobile browsers. Our extensive experimental analysis shows that OTC introduces a latency of less than 6 ms when compared to cookies - a negligible overhead for most web applications. Moreover, we show that OTC can be combined with HTTPS to effectively add another layer of security to web applications. In so doing, we demonstrate that One-Time Cookies can significantly improve the security of web applications with minimal impact on performance and scalability.

1 Introduction

HTTP is a stateless protocol. Requests to a web server are treated as independent transactions with no relation to each other. While simple and scalable, this design is not adequate for web applications that require sessions - the association of multiple transactions to a single user (e.g., online banking and e-commerce applications). HTTP cookies, small pieces of data that keep session state information in the browser,

were designed to address this limitation and rapidly became the dominant mechanism for HTTP session management.

Although cookies are a practical and efficient mechanism for session management, they introduce a number of security risks, especially when employed as session authentication tokens - a function for which they were not specifically designed [24]. For example, most web applications rely on the security provided by HTTPS to protect the user's password during the login process. During this step, the web application generates cookies that the user can later employ as lightweight session authentication tokens. However, due to performance concerns, many web applications switch to HTTP after the user logs in and cookies are transmitted "in the clear". As a result, cookies are exposed to any adversary eavesdropping on the communication. Because cookies are static, an adversary can use them to gain unauthorized access to the user's session. While these session hijacking or "sidejacking" attacks are not new, a significant number of web applications are still vulnerable [57, 58]. Several factors such as the proliferation of open wireless networks and the release of automated attack tools [29, 11, 47, 35] have increased the risk of this threat. The most recommended defense is to use HTTPS to protect all communications with the web application ("always-on HTTPS"). However, deploying always-on HTTPS can be challenging due to performance and financial concerns, particularly for distributed systems. More importantly, always-on HTTPS is not a complete solution; cookies can still be exposed due to configuration errors [39] or by attacks against HTTPS [19] and the browser [26]. In short, always-on HTTPS does not address the root cause of the problem: *cookies are weak session authenticators.*

More robust alternatives to authentication cookies have been proposed [46, 9, 5, 14]. However, they have not been adopted due to their additional requirements and complexity. Specifically, most of these alternatives require state in the web server. This is a problem for highly distributed web applications because this state needs to be synchronized among servers in different geographic locations. Thus, the effect of network latency will not only make synchronization operations more expensive, but will also cause valid requests to be denied due to "out-of-sync" state. Web 2.0 applications are particularly affected by this problem due to their higher request concurrency. In short, *proposed alternatives to authentication cookies fail to address the operational requirements of highly distributed Web 2.0 applications and, as result, have not been deployed.*

In this paper, we present One-Time Cookies (OTC), a more secure alternative to authentication cookies that does not require state in the web application. Instead of using a single, static token to authenticate each request, OTC generates a unique token per request based on a session key. Each OTC token is tied to a particular request by using a Hash-based Message Authentication Code (HMAC); hence, an adversary cannot reuse OTC tokens to illicitly redirect a session. To avoid state in the web application, OTC borrows the concept of Kerberos service tickets [45]. Like in Kerberos, an OTC session ticket contains the information the web application needs to verify an OTC token (i.e., session key), encrypted with a master key only

known by the web application. Thus, any web application's server can verify OTC tokens without keeping any volatile data, *one of the main barriers for deploying alternatives to cookies in highly distributed systems.* Unlike cookies, OTC credentials are also securely stored and isolated from other browser components. We evaluate our proposed mechanism and demonstrate an overhead similar to the insecure traditional cookie approach. In summary, *OTC preserves the performance and scalability benefits of cookies while providing stronger security guarantees.* In so doing, we make the following contributions:

- **Designing and implementing a more secure and stateless alternative to authentication cookies:** We identify key properties required to achieve a robust and practical alternative to authentication cookies. Based on these properties, we develop a protocol that prevents adversaries from successfully replaying captured authentication tokens to gain unauthorized control of a web session. Most importantly, our protocol does not require expensive state synchronization across the web application unlike previously proposed mechanisms, making it appropriate for highly distributed Web 2.0 applications. We implemented a proof-of-concept plugin for the popular blogging platform WordPress and extensions for both Firefox and Firefox for mobile browsers, demonstrating the deployability of our solution. However, we ultimately envision such mechanisms being included in the browser itself.

- **Conducting an extensive analysis on multiple platforms:** We perform extensive performance tests based on our OTC implementation, including desktop and mobile clients. Our experiments show that OTC and cookies have similar performance in both the web application and the browser. For example, the overall latency added by OTC to a WordPress page request was less than 6 ms when compared to cookies. We also apply ProVerif [7, 8] to formally verify the security properties of the OTC protocol.

- **Making our OTC implementation available to the community:** The WordPress OTC plugin and the OTC extensions for Firefox and Firefox mobile are already available online at: `http://www.cc.gatech.edu/~idacosta/otc/`. Any WordPress-based web site can incorporate OTC in matter of minutes and point their users to either the desktop or mobile Firefox extensions.

We strongly believe that OTC raises the bar against real threats, but are careful not to over claim the guarantees that OTC can provide. Specifically, while our approach efficiently eliminates session hijacking attacks by ensuring session integrity (i.e., the integrity of navigation requests), it does not provide confidentiality or full integrity protection for the information exchanged between the browser and the web application. If these additional security guarantees are required, OTC can be used together with always-on HTTPS; *OTC and HTTPS are complementary security mechanisms.* OTC's main goal is to replace cookies as session authenticators, with a performance-conscious solution that can be deployed across traditional and highly distributed web applications.

The remainder of this paper is organized as follows: Section 2 provides an overview of important related work; Section 3 offers important background information on session management on the web and presents our motivation; Section 4 explains the design and formal description of OTC; Section 5 details a security analysis of OTC; Section 6 presents our experimental testbed, tests and results; Section 7 offers additional analysis and discussion of our proposed solution and Section 8 provides concluding remarks.

2 Related Work

The use of cookies as session authentication tokens has raised security concerns since their adoption in the mid-90's. Several surveys [24, 58] have demonstrated the multiple problems with web authentication mechanisms, including vulnerability to session hijacking attacks. As a result, security researchers have proposed changes to improve the robustness of authentication cookies. Park et al. [46] and Fu et al. [24] suggested cookie mechanisms that provide better confidentiality and integrity guarantees by using well-known cryptographic techniques. In addition, these authors proposed the use of cookie expiration times to reduce the impact of session hijacking attacks. However, many applications use long expiration times to avoid affecting user experience, reducing the effectiveness of this approach. Juels et al. [34] proposed the use of cache cookies, different forms of persistent state in the browser (e.g., browser history, temporary internet files), as an alternative to cookies for storing user and session identifiers. While resistant to pharming attacks, cache cookies still need HTTPS protection to prevent active attacks. Bortz et al. [10] demonstrated a new class of attacks to steal cookies, related-domain attacks, where cookies stored by one site can be modified by another if the two sites happen to share a sufficiently long suffix. To prevent this type of attacks the authors proposed origin cookies, an extension to standard cookies that require minimal implementation costs. However, as the previous solutions, origin cookies are still vulnerable to session hijacking. Other proposed alternative to authentication cookies is the use of hidden form fields to store authentication tokens. For example, the ASP.NET ViewState [42] functionality uses this technique. However, the only difference between cookies and ViewState values are the place where they are stored in the browser; both are static tokens. Hence, ViewState is also vulnerable to session hijacking.

The problem with the previous mechanisms is that they still rely on static tokens to authenticate requests. If compromised, an adversary can reuse these tokens to send arbitrary requests. Thus, security researchers have also explored the use of dynamic authentication tokens to prevent arbitrary reuse if captured. Liu et al. [40] proposed a secure cookie protocol that creates unique cookies based on a session secret and the SSL/TLS session key. Similarly to our approach, Liu's protocol has low server state requirements; however, it assumes that secure cookies are always transmitted over HTTPS. In addition, browsers and web servers typically do not have access to SSL/TLS session keys, especially when HTTPS reverse proxies are used.

Blundo et al. [9] designed a lightweight mechanism for web caching authentication that relies on unique tokens based on a hash chain. While the use of a hash chain prevents session hijacking attacks, it requires additional state in the web application – an expensive requirement for highly distributed web applications (see Section 3.4). Ben Adida proposed SessionLock [5], a session authentication protocol that also relies on a session secret to generate unique authentication tokens to prevent session hijacking. SessionLock's main novelty is that it relies on URL fragment identifiers to store the session secret; thus, it can be implemented using only JavaScript and does not require browser modifications. However, this JavaScript-only approach makes SessionLock vulnerable to active attacks (e.g., code injection) and affects its correctness (e.g., session secrets can be exposed or lost accidentally). Moreover, SessionLock is also stateful in the web application. SessionLock is the closest work to our proposed protocol, thus, we provide a more detailed comparison in Section 7.5. More recently, Choi and Gouda presented HTTPI [14], a new secure transport protocol, more efficient than HTTPS because it only provides server authentication and session integrity (i.e., no confidentiality). As previously described solutions, HTTPI uses a session secret to generate unique authentication tokens per request. However, it also requires state in the web application. Finally, researchers have also explored the use of capability-based web servers such as Waterken [15] to provide stronger security guarantees to web applications, including session authentication. While this approach is more robust and does not require browser modifications, it still needs significant changes of the web application (i.e., new web server platform).

None of the mechanisms previously described has been widely deployed. While several of them prevent session hijacking, they fail to address the requirements of highly distributed web applications, particularly requests' statelessness. As a result, most web applications have chosen always-on HTTPS as the main defense against session hijacking attacks. To enforce always-on HTTPS and prevent downgrading attacks, Jackson and Barth [32] proposed ForceHTTPS, a browser add-on that ensures that all session cookies are securely configured and forces all HTTPS errors to be treated as critical. A similar approach is used by the Electronic Frontier Foundation (EFF) tool HTTPS Everywhere [21]. Based on this idea, a new web security policy mechanism, HTTP Strict Transport Security (HSTS), is being proposed to the IETF [31]. However, even with always-on HTTPS, cookies can be disclosed through many different attack vectors (see Section 3.3). Therefore, to effectively prevent session hijacking attacks, a more robust, efficient and practical alternative to authentication cookies is needed.

3 Web Session Authentication

We present a brief overview of HTTP cookies and their role in web session authentication. We focus on the security challenges associated with this mechanism.

3.1 HTTP Cookies

HTTP does not provide support for session management. Each request and response exchanged between a client and a server are considered an independent transaction. While this is sufficient for most basic static pages, the need for session management mechanisms increased with the development of the first web applications. In 1994, Netscape proposed the use of HTTP cookies [37, 38, 6] for web session management. Due to their simplicity and efficiency, HTTP cookies were rapidly adopted by all major browsers and web applications, becoming the default mechanism for web session management.

Cookies consist of name-value pairs containing session information that is stored in the browser. A web application generates cookies and sends them to the browser using the *Set-Cookie* HTTP response header field. In addition to the cookie's name and value, the web application can define other attributes such as the *domain* and *path* to define the cookie's scope, the *expiration* to determine cookie's lifetime, the *httponly* flag to decide if client-side scripts can access the cookie and the *secure* flag to determine if the cookie should be transmitted only using a secure channel (i.e., HTTPS). Once cookies are accepted by and stored in the browser, they are appended to each request sent to the web application, using the *Cookie* HTTP request header field.

3.2 Cookies as Session Authentication Tokens

Authentication cookies are generally created during the user login process. After successful validation of the user's authentication credentials, the web application generates authentication cookies and sends them to the browser. The browser attaches these cookies to each request that requires authentication, based on the cookies' scope and flags. Once established, authentication cookies become a temporary replacement of the user's password credentials.

Authentication cookies should be carefully constructed to prevent abuse. However, due to the heterogeneity of web applications, there are no standards for designing and implementing cookie-based session authentication mechanisms. As a result, many web developers design and implement in-house mechanisms, frequently introducing critical vulnerabilities [24, 58]. In general, web applications rely on well-known cryptographic techniques (e.g., symmetric encryption algorithms and cryptographic hash functions) and secret information (e.g., cryptographic keys, users' passwords) to build authentication cookies. The secret information is shared among all the web application's servers that need to verify the authenticity of users' requests. Nevertheless, while the confidentiality and integrity of cookies can be guaranteed by cryptographic mechanisms, attacks are still possible based on how cookies are used.

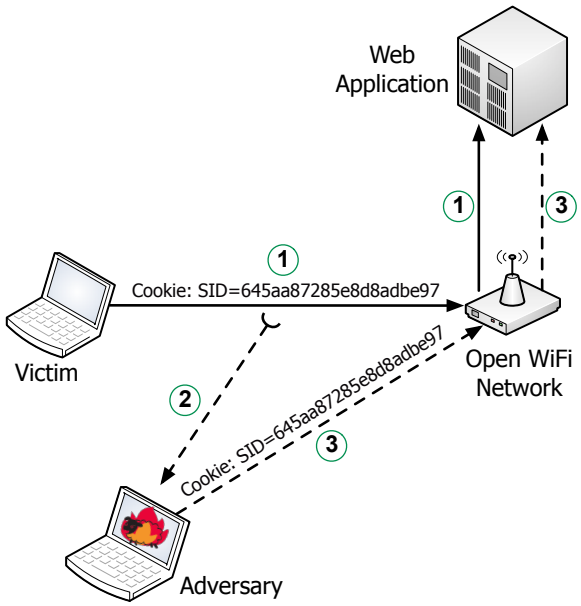


Figure 1: Simplified view of a session hijacking attack. (1) After login, the victim sends requests to the web application using a cookie for authentication. (2) Because this request is sent over HTTP, an adversary can eavesdrop the request and capture the cookie. (3) Finally, the adversary can use this cookie to send arbitrary requests to the web application, successfully hijacking the victim's session.

3.3 The Session Hijacking Threat

By design, cookies are static; they do not change during their lifetime. Hence, if an adversary steals authentication cookies, she will be able to impersonate the user associated with these cookies. This type of attack is known as *session hijacking* or *sidejacking* because the adversary takes control over the user's session.

Figure 1 shows a simplified view of a session hijacking attack. After logging in, the victim uses an authentication cookie (*SID*) on each request to the web application (step 1). As it commonly happens, the cookie is sent unprotected across the network and it is captured by an adversary eavesdropping on the communication (step 2). For example, the adversary can use an automated tool such as FireSheep [11] for this attack. Finally, the adversary can use the stolen authentication cookie to make arbitrary requests to the web application as the user (step 3), until the cookie expires. It is important to note that the stolen cookie will remain valid even if the victim logs out from the web application. Cookies are stateless; therefore, the web application cannot revoke them (the web application could change the key(s) used to create the cookies, but that will revoke the cookies for all users).

Session hijacking attacks are not new; however, several factors have increased the risk of this threat. First, the increasing popularity and importance of web applications makes them a valuable target. Google, for example, was forced to improve the security of Gmail because of several incidents against its users in China [55, 12]. Second, the proliferation of wireless networks, particularly open Wi-Fi access points (e.g., airports, libraries, stores) increases the risk of these types of attacks. Third, the release of several automated,

easy-to-use tools to perform session hijacking [29, 11, 47, 35] has brought session hijacking to the masses. For example, FireSheep [11], the most popular of such tools, has been downloaded almost 2 million times since its release in December 2010.

Cookies include an expiration time to reduce the window of opportunity for a session hijacking attack. However, many web applications use long expiration times (e.g., from hours to weeks if the "remember me" option is used) to avoid affecting user experience. This approach reduces the effectiveness of the expiration time against session hijacking. Cookies could also employ other alternatives to guarantee their freshness such as nonces (e.g., challenge-response protocol) or counters (e.g., one-time password mechanisms such as HOTP [44]). While these mechanisms are more robust against session hijacking than timestamps (i.e., expiration time), they require additional messages per request or additional state in the web application.

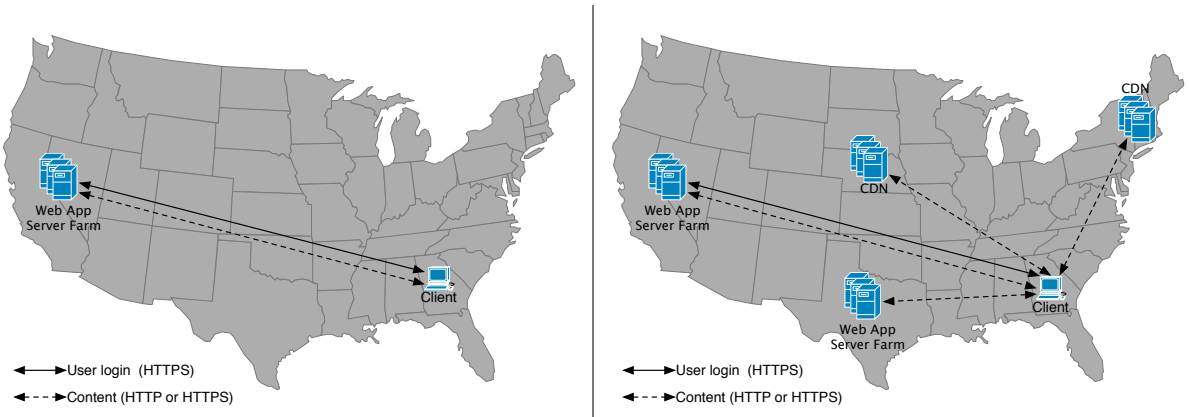
Always-on HTTPS is the most recommended defense against session hijacking. However, always-on HTTPS may be difficult to deploy, particularly in large web applications not originally designed for such requirement. Always-on HTTPS not only affects performance (e.g., additional cryptographic overhead, web-caching mechanisms do not work with HTTPS) but also impacts existing functionality (e.g., virtual hosting, applications [27], network content filtering [49]). In addition, even with always-on HTTPS, authentication cookies can be exposed accidentally [25, 18, 39] or stolen by attacking HTTPS [48, 19, 13, 54]. Moreover, HTTPS only protects cookies on the network. An adversary can also steal cookies from the user's computer through many different attacks (e.g., cross-site scripting [33], cross-site tracing [30] and related-domain [10] attacks).

In summary, the simple design of cookies makes them vulnerable to session hijacking. Although authentication cookies are a shared secret between the browser and the web application, they are treated as standard cookies and can be easily disclosed. Consequently, additional protection mechanisms are required to safeguard authentication cookies while traveling on the network and while stored in the browser. This additional protection adds complexity to the security architecture of web applications.

3.4 State Management in Large Web Applications

Cookies are preferred for session authentication mainly because they are a simple and stateless mechanism. A web application's server only needs to know the secret key(s) used to create the cookie in order to verify it. This key(s) is typically configured during the server setup and has a long lifetime. The statelessness of the client-server requests is a critical requirement for the performance and scalability of highly distributed Web 2.0 applications. For example, statelessness is one of the six constraints of the Representational state transfer (REST) software architecture model [23], adopted by service providers such as Yahoo, Google and Facebook [52].

Figure 2a shows a simplified view of the traditional model used by web applications. Clients located



(a) Traditional model

(b) New model

Figure 2: Simplified view of the traditional and the new Web 2.0 client-server models. In the new model, state synchronization among the web application's servers is expensive due to network latency. Therefore, requests should be stateless, including their authentication information.

in different geographical areas send requests and retrieve content from a set of web application's servers located in a single physical location. In this model, synchronizing state among the servers is typically not expensive. However, highly distributed Web 2.0 applications have replaced the traditional model with the one depicted in Figure 2b. In the new model, a web application has data centers in diverse locations and relies on Content Delivery Networks (CDN). Clients send requests and retrieve content from servers located in different geographical areas, typically closer to the client. This approach provides several benefits such as better redundancy, improved access bandwidth and reduced access latency. However, in this model synchronizing state among all the web application's servers is expensive. For example, the web application cannot guarantee that all the servers have the same state information at a given moment. If this state information is required for authenticating requests, then some requests are likely to be wrongly denied.

Unfortunately, proposed alternatives to authentication cookies fail to address the statelessness requirements (see Section 2). While robust against session hijacking, the proposed mechanisms require additional state in the web application [46, 9, 5, 14]. As a result, these alternatives have not been seriously considered for deployment and authentication cookies remain in use.

4 One-Time Cookies: A Robust and Stateless Session Authentication Protocol

We propose an alternative mechanism to replace cookies as session authentication tokens. Our solution, One-Time Cookies (OTC), provides robust defense against session hijacking while complying with the requirements of highly distributed applications. In particular, OTC separates session authentication from other session management tasks.

4.1 Threat model

In our model, the adversary's goal is to take control of sessions already established by users of a web application. OTC assumes two types of polynomial-time adversaries (PPT): passive and active. A *passive adversary* has access to all the information exchanged between the browser and the web application. She can access this information directly from the network (online) or from network logs (offline). Based on this information, the passive adversary will try to fabricate or reuse authentication tokens to hijack a user's session. An *active adversary* has the same access to information as the passive one, but in addition, this adversary can actively modify the requests and responses exchanged between the browser and the web application. For example, the active adversary can modify, create and prevent messages from reaching their destination. In addition, an active adversary can execute application level attacks against the browser and the web application, including cross-site scripting (XSS), cross-site tracing (XST) and session fixation attacks. An active adversary can also try phishing attacks to steal OTC tokens or try to steal the OTC's persistent storage file from the user's computer. We do not consider attacks where the adversary takes control of the user's browser or OS (e.g., by exploiting a buffer overflow or through malware) or attacks that compromise the web application infrastructure. Moreover, OTC does not defend against denial of service attacks.

OTC relies on HTTPS to protect the setup of its credentials during the user login. Therefore, OTC assumes that HTTPS is established correctly and in a secure way. We do not consider attacks that break the confidentiality guarantees offered by HTTPS during user login. If such attacks were possible, the adversary could also steal the user's password - a more valuable credential. Still, we do consider attacks against HTTPS connections established after user login.

4.2 Desired Protocol Properties

We identified properties required to achieve a robust and practical alternative to authentication cookies. We then used these properties to design OTC:

- *Session Integrity*: the proposed mechanism should provide robust client-side session authentication and it should be inherently secure against session hijacking (i.e., no additional protection mechanisms should be required).

- *Statelessness*: the proposed mechanism should not require additional state in the web application for request verification. In other words, it should not be different from authentication cookies in terms of server state requirements. As described in Section 3.4, this property is critical for highly distributed web applications.

- *Robustness*: the proposed mechanism should generate authentication tokens with strong confidentiality and integrity guarantees. In particular, authentication tokens should not leak information that compromises the security of the web application, should be resistant to cryptanalysis attacks (e.g., volume attacks) and should be tamper-evident.

- *Performance and Scalability*: the proposed mechanism should be as efficient and scalable as authentication cookies. Web application's performance and scalability should not be affected.

- *Secure storage*: the proposed mechanism should store authentication credentials securely in the browser. In particular, authentication credentials should be isolated from other browser components and functionality. For example, credentials should have similar protection as passwords and private keys. Any persistent storage should be protected with encryption.

- *Deployability*: the proposed mechanism should require minimal changes in the browser and the web application. No additional hardware or software should be required. In addition, it should be easy to configure in both the browser and the web application.

- *Usability*: the proposed mechanism should provide a similar user experience to cookies. No additional user interaction should be required. In general, the user experience should not change after upgrading from authentication cookies to OTC.

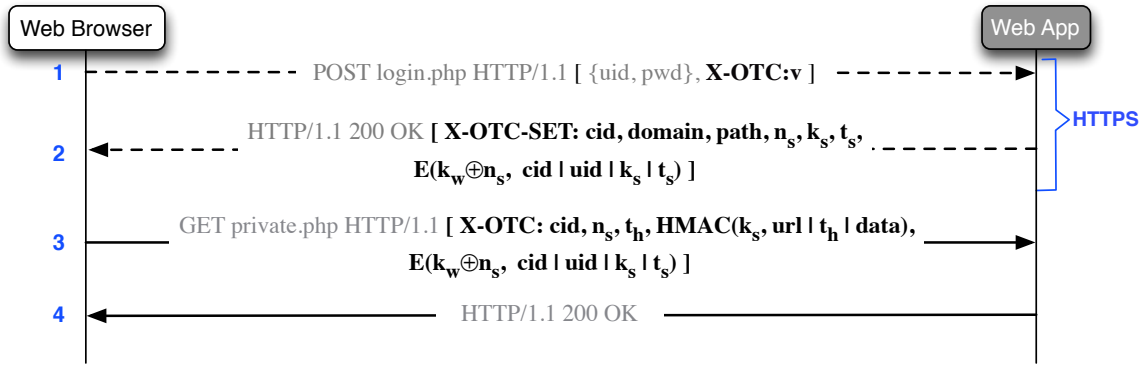
- *Concurrency*: the proposed mechanism should work with web applications that have high request concurrency (e.g., AJAX). Thus, authentication tokens should be independent of each other (i.e., avoid serialization).

- *Browser support*: The proposed mechanism should be implemented as part of the browser (core component or extension) to provide adequate security and functionality guarantees. This property is important because the mechanism requires access to every HTTP request the browser sends to the web application.

4.3 Protocol Description

OTC creates a unique token per request. Each token is bound to a particular request by using a session secret; thus, a token cannot be reused for different requests. In addition, OTC borrows the concept of tickets from Kerberos [45] to store the state information required to validate the token. Each ticket is encrypted with a long-term key shared among all the web application's servers (k_w).[1] Hence, only the web application's servers can access the information stored in the ticket. The user never has access to the contents of this

[1]Recall that distributed web servers already often share a single key for validating traditional authentication cookies, so we are not requiring any additional state.



uid, pwd	:	user ID and password	cid	:	OTC credential's ID
k_s, n_s	:	session key, session nonce	*domain, path*	:	OTC credential's scope
k_w	:	web application long-term key	E(k, x)	:	symmetric encryption of x with key k
t_s, t_h	:	session and token expiration times	HMAC(x)	:	cryptographic hash-based message authentication code of x
url	:	url of the requested resource			
$data$	:	POST form data	$X\text{-}OTC$	:	HTTP header fields for exchanging OTC protocol information
v	:	OTC protocol version			

Figure 3: Flow diagram of a web session using OTC. Messages 1 and 2 represent the user login transaction and require HTTPS protection. After user login, HTTPS is optional; each browser request includes a unique OTC token (message 3) to authenticate the request.

ticket. We define *credentials* as the values stored in the browser and *tokens* as the values attached to each request. OTC tokens are created based on the OTC credentials stored in the browser.

Figure 3 shows how OTC credentials are established and used. During user login (message 1), the browser sends the user's ID (uid) and password (pwd) to the web application. In addition, the browser includes a special HTTP header field: $X\text{-}OTC$. This header field indicates that the browser supports OTC session authentication and the OTC protocol version (v). After successful user authentication, the web application checks if the $X\text{-}OTC$ header field is present in the request. If the field is present, the web application generates OTC credentials for the newly created session. The OTC credentials consist of the credentials' ID (cid), credentials' scope ($domain$ and $path$), a session nonce (n_s), a session key (k_s), a session expiration time (t_s) and a session ticket ($E(k_w \oplus n_s, cid|uid|k_s|t_s)$). The cid, $domain$ and $path$ parameters are used in scenarios where the web application requires more than one set of credentials per user. For example, the web application may require one set of OTC credentials for basic operations and another set for administrative operations (see Section 6 for an example). If the $X\text{-}OTC$ header field is not present in the browser request, the web application could switch to standard authentication cookies or halt the communication and notify the user that OTC support is mandatory.

The web application sends the OTC credentials to the browser (message 2) using a special HTTP header

field: *X-OTC-SET*. The credentials are sent over the same HTTPS channel used to protect the user's password. Once received, the browser stores the credentials in protected storage, isolated from other browser components.

On every request that matches the credential's scope, the browser attaches an OTC token using the *X-OTC* header field (message 3). The OTC token consists of the credential's ID (cid), the session nonce (n_s), the token's expiration time (t_h), a hash-based message authentication code (HMAC($k_s, url|t_h|data$)) and the corresponding session ticket ($E(k_w \oplus n_s, cid|uid|k_s|t_s)$). The HMAC computation includes the request's URL (url), the token's expiration time (t_h) and any web form information ($data$) included in POST requests (GET requests' parameters are included in the URL). The OTC token is stateless; *the ticket contains all the information required by the web application to validate the HMAC* (statelessness property, Section 4.2). In addition, OTC-tokens are self-contained; thus, they can be verified independently. This property guarantees that OTC tokens can be used in web applications with high concurrency (e.g., AJAX) (concurrency property, Section 4.2).

After receiving the request, the web application validates it using the attached OTC token. First, the web application verifies that the token has not expired (i.e., checks t_h). Then, it uses the long-term key (k_w) and the session nonce (n_s) to decrypt the session ticket. If the decryption is successful, the web application validates that the ticket has not expired (i.e., checks t_s) and that the credentials' ID (cid) and user's ID (uid) belong to the current session. Next, the web application computes a new HMAC using the session key k_s and the information in the request (*url*, *data*). It then compares the newly computed HMAC with the HMAC included in the OTC token. If the values match, the request is accepted and the web application returns the requested resource with a *200 OK* HTTP status code (message 4). If the HMAC values do not match or if any of the previous checks fail, the request is denied and the web application redirects the browser to the login page.

The session continues until the session ticket expires (based on t_s) or the user explicitly logs off. To log off, the browsers send a request to the web application with its corresponding OTC token (not shown in Figure 3). The web application verifies the token and sends back a new *X-OTC-SET* header that only includes an HMAC of the value zero (0) using the session key ($HMAC(k_s, 0)$). This HMAC indicates the browser to delete the OTC credentials for this domain (browser enforced policy). By including this HMAC value, OTC prevents the arbitrary deletion or modification of the OTC credentials via spoofed server responses.

5 OTC Security Analysis

5.1 Informal Analysis

With OTC, the browser signs each request with the session secret k_s, instead of attaching it to the requests (as cookies do). Therefore, the browser never sends the session secret across the network, reducing its exposure. The session secret is only transmitted over the network when the web application sends it to the browser during user login (message 2 in Figure 3). However, HTTPS is used to protect the session secret during this step. As stated before, OTC assumes that HTTPS is correctly established during user login, otherwise an adversary can also learn the user's password.

OTC relies on an HMAC function to sign users' requests. The inclusion of the token's expiration time (t_h) guarantees that each HMAC value is unique, even for identical requests. Thus, the HMAC makes each OTC token unique and ties each token to a particular request. As a result, an adversary (active or passive) will not be able to reuse captured OTC tokens for arbitrary requests (session integrity property, Section 4.2). In addition, an adversary cannot fabricate or modify OTC tokens because she does not know the session secret (k_s).

An active adversary could still try to resend a previously observed request (i.e., replay attack). However, the adversary is limited to replay exactly the same request; she cannot modify the request's payload because it is protected by the HMAC. To make this attack even more difficult, OTC tokens also include an expiration time (t_h). The token's expiration time should have a shorter duration than the session expiration time (t_s). For example, $t_s = 1$ hour and $t_h = 30$ seconds. This approach significantly reduces the window of opportunity for a replay attack. To avoid time synchronization problems, both t_s and t_h should be computed based on the web application's clock.

To hijack a session using OTC, an adversary needs to learn k_w or k_s. We assume that k_w is securely protected by the web application. Thus, it is more likely that an adversary will try to learn k_s by stealing the OTC credentials from the user's browser - the only place where k_s is stored. However, OTC credentials are isolated from other browser components by default. In addition, OTC's persistent storage is protected by encryption. Therefore, none of the known cookie-theft attacks are likely to succeed in stealing OTC credentials (secure storage and browser support properties, Section 4.2)

The confidentiality and integrity of OTC tokens (robustness property, Section 4.2) are guaranteed by well-known cryptographic constructs: symmetric encryption algorithms (e.g., AES) and cryptographic hash functions (e.g., MD5, SHA-1). OTC tokens do not leak information; sensitive information about the web application is only included in the session ticket. To increase the difficulty of cryptanalysis attacks, the session tickets are encrypted with a salted version of k_w (i.e., $k_w \oplus n_s$). Thus, by using the session nonce n_s as a salt for k_w, each session ticket is always encrypted with a different key.

Threats on the network	**Cookies**	**OTC**
Disclosure due to use of unencrypted HTTP	x	-
Disclosure due to configuration errors/software bugs [25, 18, 39, 28]	x	-
SSL splitting attacks [41, 48]	x	-
SSL renegotiation attacks [19]	x	-
SSL BEAST attacks [54]	x	-
Denial of service attacks	x	x
Threats on the browser and web application	**Cookies**	**OTC**
Cross-Site Scripting (XSS) attacks [56, 33]	x	-
Cross-Site Tracing (XST) attacks [30]	x	-
Cross-Site Request Forgery (CSRF)	x	x
Related-domain attacks [10]	x	-
Clickjacking attacks [50]	x	-
Session fixation attacks [36]	x	-
"Protected" cookie clobbering [60]	x	-
Weak token generation [24]	x	-
Cookie-stealing malware [59]	x	-
Malware controlling the browser	x	x
Social engineering attacks	x	x

Table 1: Main threats affecting authentication cookies. OTC is robust against most of these threats; thus, it effectively reduces the attack surface affecting session authentication based on cookies and simplifies the security architecture of the web application. (Note: x = affected by the threat, - = not affected by the threat).

In contrast to authentication cookies, OTC requires a signed response message from the server to delete or modify existing OTC credentials in the browser. After a log off request, the server responds with a signed value using the existing session key (k_s). This approach prevents session fixation attacks [36] and "protected" cookie clobbering [60] that take advantage of the fact that cookies can be overwritten by spoofed server responses or malicious JavaScript code.

Compared to cookies, OTC requires a simpler security configuration. OTC does not require additional mechanisms to protect against session hijacking. For example, OTC does not require the *httponly* and *secure* flags, which are often misunderstood or ignored by web developers [61]. Also, OTC does not require always-on HTTPS to prevent session hijacking; thus, providing an alternative to web applications that cannot deploy always-on HTTPS.

Table 1 shows a list of the main threats affecting authentication cookies and if they apply to OTC. Except for denial of service attacks, network attacks do not affect OTC because the browser never sends the OTC session secret across the network. In the browser, OTC is resilient to most attacks affecting cookies because OTC credentials are securely stored and managed in the browser by default. Only attacks where the adversary takes control of the browser (e.g., CSRF, malware) can also affect OTC. Therefore, *OTC significantly reduces the attack surface affecting session authentication on web applications.*

5.2 ProVerif Analysis

OTC tokens do not leak information that could allow an attacker to learn the session key k_s or the web application's long-term key k_w. To verify this property more formally, we used ProVerif [7, 8], a tool for automatically analyzing the security of cryptographic protocols in the formal adversarial model (i.e. Dolev-Yao model [20]). For this purpose, we modeled the OTC protocol (Figure 3) using *pi calculus*. Using this model and ProVerif, we successfully proved the following OTC's security properties:

- *Secrecy of* k_s: the value k_s is only known to the browser and the web application.
- *Secrecy of* k_w: the value of k_w is only known to the web application.
- *Authentication of the browser to web application*: if the web application reaches the end of the protocol and believes it has shared the k_s with the browser, then the browser was indeed its interlocutor and it has shared k_s.

To test secrecy, ProVerif verified the reachability properties of k_s and k_w based on our model. To test authentication, ProVerif verified correspondence assertions between the two events: when the browser accepted k_s and when the web application finished validating an OTC token. In our analysis, the symmetric encryption algorithm is assumed to be indistinguishable under chosen plaintext attacks (IND-CPA) and to satisfy ciphertext integrity (INT-CTXT). In addition, the HMAC scheme is assumed to be unforgeable under chosen message attacks (UF-CMA). The OTC model in pi calculus and the output results from ProVerif are shown in the Appendix.

These results confirm that, in order to break the secrecy and authentication properties of OTC, an adversary will have to break the security of the underlying cryptographic components: symmetric encryption, HMAC and cryptographic hash functions. Therefore, by observing and/or modifying the communication between the browser and the web application, the adversary gains little advantage against OTC (robustness property, Section 4.2).

6 Experimental Evaluation

6.1 OTC Implementation

We implemented OTC's browser and web application components for our experimental evaluation. In the web application, we added OTC support to WordPress v.3.2.1 [3], one of the most popular open-source web content management system on the Internet. In addition, we configured WordPress with the BuddyPress plugin v.1.5.1 [1] to add more Web 2.0 and social networking functionalities to WordPress. OTC was implemented as a WordPress plugin, requiring less than 200 lines of PHP code. This code replaces the

WordPress Authentication Cookies
Set-Cookie: wordpress_sec_6e7a6b34f1dd07c511f0105e2f4708a8=admin%7C1320449253%7Cf109f3bb1777a7ca8594286d18e68096; path=/wordpress/wp-content/plugins; secure; httponly **Set-Cookie:** wordpress_sec_6e7a6b34f1dd07c511f0105e2f4708a8=admin%7C1320449253%7Cf109f3bb1777a7ca8594286d18e68096; path=/wordpress/wp-admin; secure; httponly **Set-Cookie:** wordpress_logged_in_6e7a6b34f1dd07c511f0105e2f4708a8=admin%7C1320449253%7C5623496b9ed718a32810ffd056e 0d7e8; path=/wordpress/; httponly **Cookie:** wordpress_sec_6e7a6b34f1dd07c511f0105e2f4708a8=admin%7C1320449253%7Cf109f3bb1777a7ca8594286d18e68096; wordpress_logged_in_6e7a6b34f1dd07c511f0105e2f4708a8=admin%7C1320449253%7C5623496b9ed718a32810ffd056e0d7e8;
WordPress OTC Credentials and Token
X-OTC-SET: otc;.myapp.org;/;I9LBXQvMjty1XpcCEl/cvw==;m2jS5QpaaBrfA9iaV8Hqyg==;1320280134;D0ue2+HIm0sFglMJDhty OAK614mkTWEtC/angk2nQ9acG/AdeKaaF+Z+x1LZn+OU **X-OTC:** otc;I9LBXQvMjty1XpcCEl/cvw==;1320276630;w3rzD/lrPgtG4cv0EQrplg==;D0ue2+HIm0sFglMJDhtyOAK614mkTWEtC/a ngk2nQ9acG/AdeKaaF+Z+x1LZn+OU

Table 2: Example of authentication cookies and OTC credentials for WordPress. Cookies are setup in the browser with the *Set-Cookie* HTTP header field while OTC credentials are setup with the *X-OTC-SET* header field. On each request that requires authentication, the browser attaches the cookies using the *Cookie* HTTP header field or attaches an OTC token using the *X-OTC* HTTP header field. WordPress uses 3 authentication cookies by default.

creation and verification functions of authentication cookies with equivalent OTC functions. The OTC plugin can be installed using the standard WordPress administrative interface in less than 5 minutes (deployability property, Section 4.2).

In the browser, OTC was implemented as an extension for Firefox v.7.0.1 and Firefox for mobile (Fennec) v.4.03b (browser support property, Section 4.2). The OTC browser extension required approximately 300 lines of JavaScript code. This extension can be installed using Firefox add-ons interface in less than 5 minutes (deployability property, Section 4.2). Both, OTC WordPress plugin and Firefox extensions are currently available for evaluation at `http://www.cc.gatech.edu/~idacosta/otc/`. We ultimately envision OTC as being included with core browser functionality, so that all users would benefit without having to install an extension.

Table 2 shows an example of the WordPress authentication cookies and the equivalent OTC credentials based on our implementation. By default, WordPress requires three authentication cookies: two for accessing administrative operations (e.g., changing password) and one for general operations (e.g., posting a new message). The two first cookies, used for administrative tasks, have different scopes and both have the *secure* and *httponly* flags enabled. The purpose of these cookies is to limit the impact of a session hijacking attack. In contrast, OTC only requires a single set of credentials and a single token to authenticate the request because it is inherently robust against session hijacking. In other words, OTC offers simpler but

stronger session integrity protection than cookies. As cookies, OTC also allows multiple sets of credentials by using the scope parameters (i.e., domain and path); however, in most scenarios, a single set of credentials should suffice.

As Table 2 shows, OTC uses Base 64 encoding for its credentials and tokens. For a more direct comparison with WordPress cookies, OTC uses an HMAC based on the MD5 cryptographic hash function (HMAC-MD5) and AES with 128 bit keys (AES-128) for symmetric encryption. However, it is easy to configure OTC with stronger algorithms. Note that symmetric encryption operations are performed by the web application only; the browser does not need to encrypt or decrypt information (i.e., the session ticket).

6.2 Evaluation and Results

The main goal of our experimental evaluation is to characterize and compare the performance of OTC and authentication cookies. First, we measured the delay added by single OTC and cookie operations. For this purpose, we used code instrumentation in WordPress and Firefox (desktop and mobile). Second, we characterized the system-level impact on performance of OTC and cookies on WordPress. We focused on metrics such as page load times, maximum throughput and CPU utilization. All the experiments were executed at least 20 times to ensure the soundness of the results. In addition, mean values and 95% confidence intervals are reported for all the experiments. Finally, we also ran informal experiments to evaluate the usability and compatibility of OTC with WordPress.

6.2.1 Microbenchmarks

For a direct comparison with WordPress's cookies, OTC was configured to use HMAC-MD5 and AES-128. In our experiments we used a laptop (MacBook Pro with dual core 2.53 GHz processor, 4GB of memory and Mac OS X 10.6) and a smartphone (Google Nexus One with 1 GHz processor, 512 MB of memory and Android 2.3.6) as our clients. WordPress was installed in an Ubuntu v.8.04 (Linux Kernel 2.6.24) server with 2 Quad-Core 2.00 GHz processors, 16 GB of memory and Gigabit Ethernet cards. The server was also configured with WordPress's supporting software: Apache v.2.2, MySQL v.5.0 and PHP v.5.3. All the server software used a default configuration (i.e., no performance optimizations).

We first measured the time required to generate authentication cookies and OTC credentials and the time required to validate cookies and OTC tokens. Table 3 shows the average results and confidence intervals. For both generation and validation, OTC operations required more time than cookies; however, this difference is negligible when compared to other WordPress operations (performance and scalability properties, Section 4.2). For example, loading a WordPress page typically requires hundreds of milliseconds. OTC required 0.2060 ms and 0.3990 ms more than cookies for generating credentials and validating tokens, respectively. These differences are expected because OTC uses symmetric encryption operations and extra

Protocol	Generation (ms)	Verification (ms)
Cookies (95% c.i)	0.1610 (±0.0020)	1.6060 (±0.4500)
OTC (95% c.i.)	0.3670 (±0.0120)	2.0050 (±0.0610)

Table 3: WordPress generation and verification times for cookies and OTC. The additional delay added by OTC operations is small (< 1 ms) and negligible when compared to other web application's operations. Note: c.i. = confidence intervals.

Device	OTC Token Generation (ms)
Laptop (95% c.i.)	0.1335 (±0.0034)
Smartphone (95% c.i.)	2.3945 (±0.0974)

Table 4: Time to generate tokens in the browser. The overhead added is small and unlikely to affect the user experience

validation steps in addition to those used by cookies.

On the browser side, we measured the time that OTC requires to generate and attach an authentication token to each request. Table 4 shows the average results and confidence intervals for the laptop and for the smartphone. The results show that the overhead added by OTC to Firefox (desktop and mobile) is also small and unlikely to affect the user experience.

6.2.2 Macrobenchmarks

Our first macrobenchmark experiment consisted of measuring the overall latency added by OTC to WordPress's responses. For this purpose, in the browser we measured the time required to load the home page from WordPress for the following configurations: cookies with HTTP, cookies with HTTPS, OTC with HTTP and OTC with HTTPS. We measured this time only for new TCP or SSL/TLS connections (i.e., no channel reuse) using Firebug [2], a Firefox extension for web development. The WordPress home page requires 14 requests for resources (e.g., images, css files) and has a size of 210.4 KB. In addition, we added a latency of 50 ms between the browser and the web application to simulate a more realistic Internet round trip time.

Figure 4 shows the results of this experiment. For HTTP, the WordPress page required approximately 1.14 sec and 1.21 sec to load for cookies and OTC, respectively. For HTTPS, it required approximately 1.41 sec for cookies and 1.49 sec for OTC. Thus, the additional latency introduced by OTC is around 70 ms for HTTP and 80 ms for HTTPS. These values represent to the total time to load the WordPress page, which requires 14 requests. Therefore, the mean latency added by OTC to each request is approximately 5.00 ms and 5.71 ms for HTTP and HTTPS, respectively. Taking into account the margin of error of this experiment, these values resemble the ones measured in our microbenchmarks plus a small amount of network jitter. Therefore, these results confirm that the latency added by OTC to the page load time is negligible (performance and scalability properties, Section 4.2).

Our second macrobenchmark experiment measured OTC's overall impact on the maximum throughput

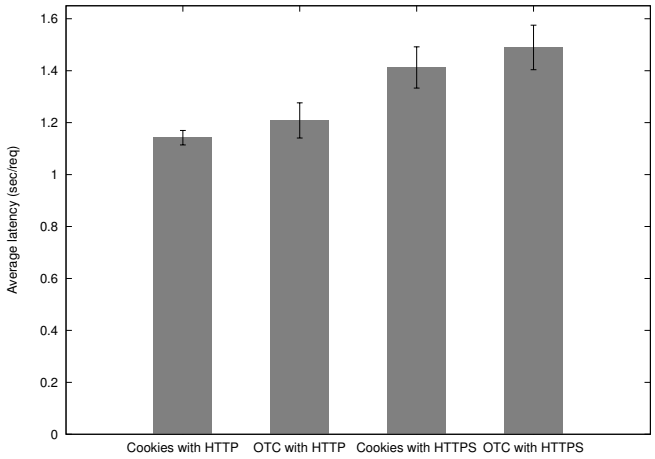


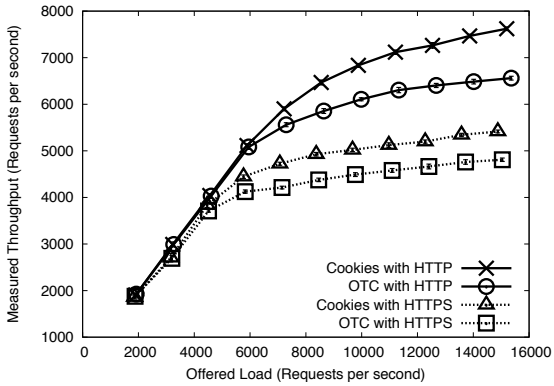
Figure 4: Average user experienced latency per request for cookies with HTTP, cookies with HTTPS, OTC with HTTP and OTC with HTTPS. When compared to cookies, the delay introduced by OTC is small and unlikely to be noticed by the user.

that the web application can support. For this purpose, we characterized the maximum throughput (requests per second) for the four configurations described in the previous experiment. For a more realistic comparison of HTTP and HTTPS configurations, we focused on measuring performance during HTTPS steady state, avoiding the costs of HTTPS connection setup (most expensive SSL/TLS operation [16]). Therefore, in these experiments we used a small and constant number of connections while increasing the number of requests made over these connections (as opposed to increasing the number of connections). In other words, we simulated the load generated by users that already logged in to the web application.

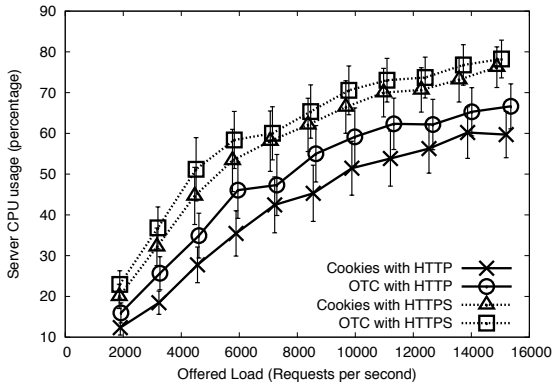
To generate the traffic load, we used *httperf 0.9* [43], a tool for measuring the performance of web servers. A total of three httperf instances (one per server) are used to generate the test loads. We wrote our own custom script to automate execution and data collection of the benchmarking experiments. These servers had a similar hardware configuration to the WordPress server. In addition, our testbed used a dedicated Gigabit Ethernet switch.

In our experiment, instead of requesting a WordPress page, the performance tool requested a small PHP page (14 Kbytes) that contained some blocks of text and OTC and cookie verification support. The reason for this setup is that WordPress pages are more complex and slower to load, resulting in a much lower throughput ($<$ 100 requests/sec). This low throughput did not allow us to measure the differences among the configurations. In short, WordPress pages become an earlier bottleneck for server performance instead of cookies or OTC.

The results for request throughput are shown in Figure 5a. As expected, the configurations using HTTP performed better than the configurations using HTTPS. When HTTP is used, the web application supported a maximum throughput of approximately 7,500 requests/sec for cookies and 6,500 request/sec for OTC. These



(a) Web server throughput



(b) Web server CPU utilization

Figure 5: Web server throughput and CPU utilization for cookies with HTTP, cookies with HTTPS, OTC with HTTP and OTC with HTTPS. These tests used a simple PHP page (14 KB) because WordPress throughput was too low (< 100 req/sec) to see performance differences among the configurations evaluated. Cookies and OTC allow similar performance in the web server for practical throughput values.

results represent a reduction in request throughput of approximately 13.33% when OTC is used instead of cookies. When HTTPS is used, the web application supported a maximum throughput of approximately 5,400 requests/sec for cookies and 4,900 requests/sec for OTC. These results represent a reduction in request throughput of approximately 9.26% when OTC is used instead of cookies (the smaller difference is due to the overhead added by HTTPS). Figure 5b shows the web server CPU utilization during the experiments for each configuration. When HTTP is used, OTC requires around 10% more CPU time than cookies due to the symmetric encryption operations. When HTTPS is used, OTC requires around 5% more CPU time than cookies. This difference is almost negligible, as Figure 5b shows, because of the HTTPS overhead. As in our previous experiment, these results show that OTC introduces a small performance overhead to the web application. While a 13.33% reduction in throughput is not negligible, note that these measurements were taken using a lightweight PHP page. Our WordPress implementation cannot support more than hundred requests per second; therefore, the overhead added by OTC is insignificant when compared to WordPress's own overhead (performance and scalability properties, Section 4.2). In addition, these results show that OTC can be used with both HTTP and HTTPS configurations. Finally, it is important to note that, as a mechanism to prevent session hijacking, HTTPS adds a considerably higher overhead than OTC.

6.2.3 Informal Usability and Compatibility Test

We also evaluated OTC's impact on user experience. We conducted a very informal user study where we asked other lab members to use our WordPress setup with both Firefox and Firefox for mobile. None of

the participants reported any difference between using cookies or OTC (usability property, Section 4.2). In addition, we thoroughly evaluated the functionality of WordPress and BuddyPress to detect any errors or compatibility problems when OTC was used. During this compatibility test, we did not find any problem with Firefox or with WordPress/BuddyPress functionality when OTC was used instead of cookies.

7 Discussion

7.1 Incrementally Deploying OTC

As our implementation shows, OTC can be easily deployed in today's web applications and browsers. For most web applications, the operations required to support OTC are not different from the ones currently used to support authentication cookies. The use of symmetric encryption by OTC could be considered the main difference. However, it is not uncommon for cookies to use symmetric encryption to protect sensitive user's session information.

The major difference between deploying OTC and cookies is in the browser. With OTC, the browser acquires an active role during session authentication by signing each user request (as opposed to just storing and attaching cookies to requests). Still, the operations required by OTC are already supported by most browsers' APIs (e.g., cryptographic hash operations, secure storage). We expect that OTC support in the browser will follow a similar adoption model as ForceHTTPS [32]: first available as a browser extension and then adopted natively by major browser vendors as an Internet standard (i.e., HSTS). As part of our future work, we plan to collaborate with vendors and groups such as the IETF to propose OTC as an Internet standard.

Most web applications can follow an incremental approach to deploy OTC. For example, web applications can enable OTC while still supporting authentication cookies. Initially, a small group of users (i.e., beta testers) can evaluate the web application functionality using OTC while standard users continue using authentication cookies. Next, the web application can enable OTC support for all users, indicating how to activate it on browsers (e.g., browser upgrade or through an extension). At this point, both OTC and cookies will be allowed for session authentication; the browser will indicate to the web application the type of protocol preferred. Thus, users that have not updated their browsers, will still be able to access the web application. Note that downgrading attacks are unlikely when OTC and cookies are both enabled because OTC support is announced over HTTPS during user login (see Section 4.3). After certain threshold is reached (e.g., percentage of users supporting OTC), the web application can deprecate the use of cookies for session authentication and rely on OTC only. Web applications with high security requirements (e.g., online banking) can combine OTC with always-on HTTPS to add another layer of security to their systems. This type of applications can follow a more aggressive approach and enable OTC directly as the only

session authentication mechanism (i.e., no transitional period). Thus, users will be required to update their browsers to use the application. Finally, OTC will not replace cookies for other session management tasks (e.g., shopping card, user preferences); cookies will still be needed for such functionality.

7.2 Extending OTC Integrity Protection

In addition to protecting the integrity of user's requests, OTC could also protect the integrity of the web application's responses. This approach can provide lightweight integrity protection in scenarios where it is difficult to deploy always-on HTTPS. For this purpose, the session key k_s could be used by the web application to sign the resources sent to the browsers (e.g., HTML code, JavaScript code, CSS code). Because the session key is included in each request, the web application can readily use it to sign the corresponding response (i.e., no delay is introduced by key retrieval operations). The OTC browser component will require only minor changes to support verification of web responses.

By signing web application's responses, OTC could detect in-flight page modifications (e.g., ISPs injecting adds, adversaries injecting malicious code). Web Tripwires [51] was proposed to detect such activity; however, it is not robust against some active adversaries because it does not rely on a shared secret. OTC could be combined with mechanisms such as Web Tripwires to provide a more robust integrity mechanism for web application's responses. We plan to explore this approach in future work.

7.3 OTC and Multi-Factor Authentication

Cookies are also used as lightweight second-factor authentication tokens. For example, mechanisms such as Yahoo's Sign-In Seal use long-lasting cookies to store a second-factor authentication token in the browser. The advantage of this approach is that the browser does not need modifications. However, some of these mechanisms are not effective against phishing attacks [53]. These cookies can also be stolen from the user's browser. Ben Adida proposed Beamauth [4], a more robust alternative based on specially crafted bookmark instead of a cookie. The use of bookmarks to store second-factor tokens offers better protection against cookie-theft attacks. But, as in the case of cookies, bookmarks were not designed for storing security-related information. Thus, OTC could provide a more robust alternative in this area. For example, a long-lasting OTC credential and its corresponding session key k_s could be used as a second-factor token. Because the browser isolates OTC credentials from other components, k_s offers better security guarantees than cookies or Beamauth as a second-factor authenticator.

7.4 OTC in Mobile Devices

Mobile devices such as smartphones and tablets are rapidly becoming the main platforms to consume Internet content. However, due to hardware constraints and lack of security maturity, these devices are frequently more vulnerable to security threats, including session hijacking. Mobile browsers and many mobile applications rely on cookies for session authentication. But, as with desktop computers, cookies can be easily exposed. However, the common wisdom is that smartphones are less vulnerable to session hijacking because they use the mobile operator network for Internet access instead of Wi-Fi. Unfortunately, this scenario is quickly changing. The popularity of mobile devices has created capacity problems for mobile operators, that have been forced to introduce data caps. As a result, smartphone users are increasingly relying on Wi-Fi networks to access the Internet [17]. In addition, mobile operators are also exploring how to reduce the load in their networks by using automatic Wi-Fi offloading mechanisms [22]. This trend significantly increases the risk of mobile devices being targeted with session hijacking attacks and other Internet threats. For this reason, we decided to also implement OTC as part of a mobile browser. As our experimental evaluation shows, OTC is lightweight enough to be used in mobile devices such as smartphones, tablets and other resource-constrained devices.

7.5 SessionLock

As mentioned in Section 2, SessionLock [5] is another proposed technique to prevent session hijacking without requiring always-on HTTPS support. As OTC, SessionLock uses a session secret to sign each request sent to the web application, creating unique authentication tokens per request (prior to SessionLock, Liu et al. [40] and Blundo et al. [9] have also explored this approach). The main novelty of SessionLock is that it uses URL fragment identifiers to store the session secret in the browser and employs client-side JavaScript to sign requests with this secret. Thus, SessionLock does not require browser modifications for its deployment. However, this approach has several limitations. First, URL fragment identifiers were not designed for storing browser information – this technique is an ad hoc use of this value. Second, in order to sign every possible request, SessionLock needs to rewrite every link on each web page displayed to the user. This operation can be computationally expensive for complex web pages and will not work with binary objects (e.g., Flash). This technique will also fail if the user types the URL or opens a new browser tab to send a request to the web application. In short, SessionLock can not guarantee that each request to the web application will be signed. Third, SessionLock requires additional state on the web application for the session secret, a hard to meet requirement for highly distributed web applications (see Section 3.4). Fourth, SessionLock is vulnerable to active attacks where an adversary can use the JavaScript API to steal the session secret (e.g., by code modification or injection). Fifth, SessionLock's session secret can be accidentally leaked by the user

(e.g., by sharing a link, bookmarks) or lost during the session. In summary, while SessionLock allows easier deployment by avoiding browser modifications, its ad hoc techniques are not adequate for complex, highly distributed web applications. As a result, SessionLock has not been deployed in production systems.

In contrast, OTC avoids these problems by relying on the browser's extension API. This API allows a complete monitoring of the user's requests, independently of the web technology used, and a robust isolation from client-side JavaScript code, preventing code modification or injection attacks to steal session secrets or the accidental disclosure of such secrets. Moreover, by using session tickets, OTC does not require additional state on the web application. Thus, by requiring a small, non-disruptive change in the browser, OTC provides a more robust solution to the session hijacking threat than SessionLock.

8 Conclusion

The risks associated with the use of cookies as session authentication tokens have been known for years. More robust alternatives have been proposed to replace authentication cookies; however, they have not been deployed because they fail to meet the requirements of highly distributed Web 2.0 applications. Specifically, most of the proposed alternatives require costly state synchronization across the web application, a serious concern for distributed systems. In this paper, we presented OTC, a principle-driven secure alternative to authentication cookies. OTC is not only resistant to session hijacking, but also maintains the simplicity and performance benefits of cookies. More critically, OTC addresses the shortcomings of previously proposed solutions by removing the need for state in the web application. Moreover, OTC offers another security layer to web applications that already support always-on HTTPS by reducing the threats associated with cookies; OTC and always-on HTTPS are complementary mechanisms. We developed OTC for the popular WordPress application and demonstrated that OTC has similar performance to traditional cookies. Our future work includes plans to work with standard bodies such as the IETF and vendors to ensure that OTC is included in browsers by default.

References

[1] BuddyPress.org. `http://buddypress.org/`, 2011.

[2] Firebug. `https://getfirebug.com/`, 2011.

[3] WordPress.org. `http://wordpress.org/`, 2011.

[4] B. Adida. Beamauth: two-factor web authentication with a bookmark. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[5] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the ACM international conference on World Wide Web (WWW)*, 2008.

[6] A. Barth. RFC 6265 - HTTP State Management Mechanism. https://tools.ietf.org/html/rfc6265, 2011.

[7] B. Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. `http://www.proverif.ens.fr/`.

[8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the IEEE Workshop on Computer Security Foundations (CSFW)*, 2001.

[9] C. Blundo, S. Cimato, and R. D. Prisco. A Lightweight Approach to Authenticated Web Caching. In *Proceedings of the The Symposium on Applications and the Internet*, 2005.

[10] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *Web 2.0 Security and Privacy Workshop (W2SP)*, 2011.

[11] E. Butler. Firesheep. `http://codebutler.com/firesheep`, 2010.

[12] M. Chan. China and Google: A detailed look. `http://blogs.aljazeera.net/asia/2011/03/23/china-and-google-detailed-look`, 2011.

[13] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang. Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[14] T. Choi and M. G. Gouda. HTTPI: An HTTP with Integrity. In *in Proceedings of International Conference on Computer Communications and Networks (ICCCN)*, 2011.

[15] T. Close. Waterken Server: capability-based security for the Web. `http://waterken.sourceforge.net/`, 1999.

[16] C. Coarfa, P. Druschel, and D. S. Wallach. Performance analysis of TLS Web servers. *ACM Transactions on Computer Systems (TOCS)*, 24(1), 2006.

[17] ComScore. Smartphones and Tablets Drive Nearly 7 Percent of Total U.S. Digital Traffic. `http://www.comscore.com/Press_Events/Press_Releases/2011/10/Smartphones_and_Tablets_Drive_Nearly_7_Percent_of_Total_U.S._Digital_Traffic`, 2011.

[18] L. Constantin. XSS Attack on Twitter Subdomain Allowed for Complete Session Hijacking. `http://news.softpedia.com/news/XSS-Attack-on-Twitter-Subdomain-Allowed-Full-Session-Hijacking-148240.shtml`, 2010.

[19] T. Cross. Stealing Cookies with SSL Renegotiation. http://blogs.iss.net/archive/stealingcookieswiths.html, 2009.

[20] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, Mar. 1983.

[21] Electronic Frontier Foundation. HTTPS Everywhere. https://www.eff.org/https-everywhere, 2010.

[22] Elizabeth Woyke. Automatic Wi-Fi Offloading Coming To U.S. Carriers. http://www.forbes.com/sites/elizabethwoyke/2011/04/22/automatic-wi-fi-offloading-coming-to-u-s-carriers/, 2011.

[23] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[24] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. *USENIX Security Symposium*, 2001.

[25] E. Galperin. Microsoft Shuts off HTTPS in Hotmail for Over a Dozen Countries. https://www.eff.org/deeplinks/2011/03/microsoft-shuts-https-hotmail-over-dozen-countries, 2011.

[26] D. Goodin. Newfangled cookie attack steals/poisons website creds. http://www.theregister.co.uk/2009/11/04/website_cookie_stealing/print.html, 2009.

[27] D. Goodin. Hotmail always-on crypto breaks Microsoft's own apps. http://www.theregister.co.uk/2010/11/10/lame_hotmail_encryption/, 2010.

[28] D. Goodin. Amazon purges account hijacking threat from site. http://www.theregister.co.uk/2010/04/20/amazon_website_treat/print.html, 2011.

[29] R. Graham. SideJacking with Hamster. http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html, 2007.

[30] J. Grossman. Cross-Site Tracing (XST). http://www.cgisecurity.com/whitehat-mirror/WhitePaper_screen.pdf, 2003.

[31] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). http://tools.ietf.org/html/draft-hodges-strict-transport-sec-02, 2010.

[32] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceeding of the ACM international conference on World Wide Web (WWW)*, 2008.

[33] Jehiah. XSS - Stealing Cookies 101. http://jehiah.cz/a/xss-stealing-cookies-101, 2006.

[34] A. Juels, M. Jakobsson, and T. Jagatic. Cache cookies for browser authentication (Extended Abstract). In *IEEE Symposium on Security and Privacy*, 2006.

[35] A. Koch. DroidSheep. `http://droidsheep.de/`, 2011.

[36] M. Kolsek. Session Fixation Vulnerability in Web-based Applications. `http://www.acrossecurity.com/papers/session_fixation.pdf`, 2007.

[37] D. Kristol and L. Montulli. RFC 2109 - HTTP State Management Mechanism. http://tools.ietf.org/html/rfc2109, 1997.

[38] D. Kristol and L. Montulli. RFC 2965 - HTTP State Management Mechanism, 2000.

[39] J. Leyden. AmEx 'debug mode left site wide open', says hacker. `http://www.theregister.co.uk/2011/10/07/amex_website_security_snafu/print.html`, 2011.

[40] A. Liu, J. Kovacs, and M. Gouda. A secure cookie protocol. In *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*, 2005.

[41] M. Marlinspike. sslstrip. `http://www.thoughtcrime.org/software/sslstrip/`, 2009.

[42] S. Mitchell. Understanding ASP.NET View State. `http://msdn.microsoft.com/en-us/library/ms972976.aspx`, 2004.

[43] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[44] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. RFC 4226 - HOTP: An HMAC-Based One-Time Password Algorithm. `http://tools.ietf.org/html/rfc4226`, 2005.

[45] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). `https://tools.ietf.org/html/rfc4120`, 2005.

[46] J. S. Park and R. Sandhu. Secure Cookies on the Web. *IEEE Internet Computing*, 4:36–44, 2000.

[47] B. Ponurkiewicz. FaceNiff. `http://faceniff.ponury.net/`, 2011.

[48] M. Prandini, M. Ramilli, W. Cerroni, and F. Callegati. Splitting the HTTPS Stream to Attack Secure Web Connections. *IEEE Security and Privacy*, 8:80–84, 2010.

[49] B. Prince. Google Moves Encrypted Web Search. http://www.eweek.com/c/a/Security/Google-Moves-Encrypted-Web-Search-668624/, 2010.

[50] F. Y. Rashid. IE Flaw Lets Attackers Steal Cookies, Access User Accounts. http://www.eweek.com/c/a/Security/IE-Flaw-Lets-Attackers-Steal-Cookies-Access-User-Accounts-402503/, 2011.

[51] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proceedings of the USENIX Symposium on Network Systems Design and Implementation (NSDI)*, 2008.

[52] A. Rodriguez. RESTful Web services: The basics. https://www.ibm.com/developerworks/webservices/library/ws-restful/, 2008.

[53] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[54] B. Schneier. Man-in-the-Middle Attack Against SSL 3.0/TLS 1.0. https://www.schneier.com/blog/archives/2011/09/man-in-the-midd_4.html, 2011.

[55] M. Siegler. China Syndrome: Gmail Now Defaults To Encrypted Access. http://techcrunch.com/2010/01/13/china-hacking-gmail-secure/, 2010.

[56] The Open Web Application Security Project (OWASP). Cross-site Scripting (XSS). http://www.owasp.org/index.php/Cross-site_Scripting_2010.

[57] The Open Web Application Security Project (OWASP). OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[58] C. Visaggio. Session Management Vulnerabilities in Today's Web. *IEEE Security and Privacy*, 8:48–56, 2010.

[59] J. Wyke. What is Zeus? http://www.sophos.com/medialibrary/PDFs/technical%20papers/Sophos%20what%20is%20zeus%20tp.pdf, 2011.

[60] M. Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Part2, 2008.

[61] Y. Zhou and D. Evans. Why Aren't HTTP-only Cookies More Widely Deployed? In *Web 2.0 Security and Privacy Workshop (W2SP)*, 2010.

OTC Verification using ProVerif v.1.86

Pi Calculus modeling of OTC

```
type key.
type nonce.
type timestamp.
free HTTPS:channel [private].
free HTTP:channel.
free secret1_ks, secret2_ks, secret_kw:bitstring [private].
fun encrypt(bitstring,key): bitstring.
reduc forall m: bitstring, k: key; decrypt(encrypt(m,k),k) = m.
fun hmac(bitstring, key): bitstring.
fun xor(key, nonce):key.
(* Secrecy queries *)
query attacker(secret1_ks).
query attacker(secret2_ks).
query attacker(secret_kw).
(* Correspondance events *)
event acceptsClient(key).
event termServer(key).
(* Browser to WebApp session authentication query *)
query x:key; inj-event(termServer(x)) ==>inj-event(acceptsClient(x)).
(* Browser macro *)
let browser(uid:bitstring, passwd:bitstring) =
    out(HTTPS, (uid, passwd));
    in(HTTPS, (cid:bitstring, ns:nonce, ks:key, ts:timestamp, ticket:bitstring));
    event acceptsClient(ks);
    new url:bitstring;
    new th:timestamp;
    new data:bitstring;
    out(HTTP, (ns, th, hmac((url, th, data), ks), ticket, url, data)).
(* Web Application macro *)
let webapp(kw:key)=
    in(HTTPS, (uid:bitstring, passwd:bitstring));
    new ks:key;
    new ns:nonce;
    new ts:timestamp;
```

```
    new cid:bitstring;
    out(HTTPS, (cid, ns, ks, ts, encrypt((cid, uid, ks, ts), xor(kw,ns))));
    in(HTTP, (nsx:nonce, th:timestamp, hmacx:bitstring, ticket:bitstring,
       url:bitstring, data:bitstring));
    let (=cid, =uid, ksx:key, tsx:bitstring) = decrypt(ticket, xor(kw, nsx)) in
    out(HTTP, encrypt(secret_kw, kw));
    out(HTTP, encrypt(secret1_ks, ks));
    out(HTTP, encrypt(secret2_ks, ksx));
    let (hmacy:bitstring) = hmac((url,th,data), ksx) in
    if hmacx = hmacy then event termServer(ks).
(* Main *)
process
    new kw:key;
    new uid:bitstring;
    new passwd:bitstring;
    ( !(browser(uid, passwd)) |  !(webapp(kw)) )
```

Proverif Output

```
RESULT inj-event(termServer(x)) ==> inj-event(acceptsClient(x)) is true.
-- Query not attacker(secret_kw[])
Completing...
Starting query not attacker(secret_kw[])
RESULT not attacker(secret_kw[]) is true.
-- Query not attacker(secret2_ks[])
Completing...
Starting query not attacker(secret2_ks[])
RESULT not attacker(secret2_ks[]) is true.
-- Query not attacker(secret1_ks[])
Completing...
Starting query not attacker(secret1_ks[])
RESULT not attacker(secret1_ks[]) is true.
```