One-Time Cookies: Preventing Session Hijacking Attacks with Disposable Credentials

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

Abstract

Many web applications are vulnerable to session hijacking attacks due to the insecure use of cookies for session management. The most recommended defense against this threat is to completely replace HTTP with HTTPS. However, this approach presents several challenges (e.g., performance and compatibility concerns) and therefore, has not been widely adopted. In this paper, we propose "One-Time Cookies" (OTC), an HTTP session authentication protocol that is efficient, easy to deploy and resistant to session hijacking. OTC's security relies on the use of disposable credentials based on a modified hash chain construction. We implemented OTC as a plug-in for the popular WordPress platform and conducted extensive performance analysis using extensions developed for both Firefox and Firefox for mobile browsers. Our experiments demonstrate the ability to maintain session integrity with a throughput improvement of 51% over HTTPS and a performance approximately similar to a cookie-based approach. In so doing, we demonstrate that one-time cookies can significantly improve the security of web sessions with minimal changes to current infrastructure.

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 makes the creation of applications requiring the association of multiple transactions to a single user (e.g., banking) somewhat difficult natively. HTTP cookies, which generally contain one or a small number of short identifier strings allowing a server to associate seemingly unrelated requests, rapidly became the dominant mechanism for web session management.

Unfortunately, the use of cookies introduces a number of security risks, especially when they are employed as session authentication tokens. As an example, many websites rely on strong security mechanisms such as HTTPS (i.e., HTTP over TLS/SSL) to initially authenticate a user. During this secure session, the server generates cookies that the user can later employ as lightweight authentication tokens. However, because these tokens are static and transmitted "in the clear", an adversary able to intercept them can use these cookies to gain unauthorized access to a user's session. While such *session hijacking* attacks are not new, a significant number of web applications remains vulnerable to this threat [38]. Moreover, the availability of tools such as Hamster [14] and Firesheep [6] makes such attacks simple to execute. While the common wisdom suggests the extension of HTTPS from login to site-wide protection, this solution may break certain web application functionality [34] and impact performance. Accordingly, lightweight solutions with fewer side-effects may be more appropriate.

In this paper, we present One-Time Cookies (OTC), an efficient and robust web session authentication protocol. OTC generates single-use authentication tokens based on a modified hash chain construction. These tokens, once verified by the web application, can not be reused. Moreover, each OTC credential is tied to a specific request for a resource, meaning that an adversary can not intercept and repurpose them for illicitly redirecting a session. In so doing, we make the following contributions:

- **Design, verification and implementation of One-Time Cookies:** We develop a protocol based on a modified hash chain construction that prevents adversaries from successfully replaying captured cookies to gain unauthorized control of an HTTP session. We then use ProVerif [1, 2, 5] to formally verify the security properties of the OTC protocol. Finally, we implement our construction as an extension for the popular blogging platform WordPress, which currently relies on HTTPS for login and subsequently uses cookies for session management.

- **Extensive performance analysis on multiple platforms:** We implement plugins for both Firefox and Firefox for mobile web browsers and perform extensive performance tests. Our experiments show an increase in throughput for the server by nearly 51% over enabling site-wide SSL. We also measure the OTC protocol in scenarios in which distributed web caches are currently relied upon and parameterize our experiments with round-trip times gathered from PlanetLab [33] measurements.

- **Make our OTC implementation available to the community:** The OTC code for the WordPress plug-in and the extensions for Firefox and Firefox mobile is already available here: `http://www.cc.gatech.edu/~idacosta/otc.html`. 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 are very careful not to over-claim the security guarantees provided by one-time cookies. Specifically, while our approach efficiently eliminates session hijacking attacks by ensuring session integrity (specifically, the integrity of navigation requests), it does not provide for confidentiality or integrity of the content actually requested by users. These are fundamental tradeoffs between our work and the use of site-wide HTTPS. However, our mechanism *does* directly respond to real threats (e.g., Firesheep [6]) with a performance-conscious solution that can be deployed without the need for additional hardware. Our solution therefore lies within a spectrum of valid techniques and can immediately improve the security of users of many web applications.

The remainder of this paper is organized as follows: Section 2 offers important background information on session management on the web and presents our motivation; Section 3 describes the design, implementation and formal description of the protocol in which one-time cookies are used and its security properties; Section 4 presents our experimental testbed, experiments and results; Section 5 offers additional analysis and discussion of our proposed solution; Section 6 provides an overview of important related work; Section 7 offers concluding remarks.

2 Web Session Authentication

In this section, we present a brief overview of HTTP cookies as session authentication tokens and session hijacking attacks. We then discuss current solutions against these attacks and their lack of widespread deployment.

2.1 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 the most basic static pages, the need for session management mechanisms increased with the development of the first web applications. As a result, HTTP cookies [20,21] were first proposed in 1994 and have since become the dominant mechanism for web session management.

Cookies consist of name-value pairs containing session information. A web application running on the server generates cookies and sends them to the user in responses as HTTP headers as follows:

```
Set-Cookie:  SID=645aa87285e8d8adbe97c4f3e22
```

A web application adds one Set-Cookie header per cookie to the server's response. In addition to the cookie value, the web application can define other parameters such as *domain* and *path* (cookie's scope), *expiration* (cookie's validity period), *HttpOnly* flag (defines if the cookie can be accessed by client-side scripts) and *Secure* flag (defines if the cookie can only be sent over a secure channel; i.e., HTTPS). The browser will add cookies to each request to the web application as one HTTP header as follows:

```
Cookie:  SID=645aa87285e8d8adbe97c4f3e22
```

Cookies used to authenticate user's requests (authentication cookies) are normally created during the login process. After successful validation of the user's primary authentication credentials (i.e., username and password), the web application generates new authentication cookies and sends them to the user's browser. The browser then appends these cookies to each request that requires authentication. Using cookies for session authentication offers several advantages. First, it avoids requiring the user to enter authentication credentials for every request. Second, it requires minimal state management in the web application. This is important for web applications where state synchronization is difficult (e.g., multiple servers in different geographical locations). Third, it is a simple mechanism and is supported by all browsers.

2.2 Problems with Authentication Cookies

Once established, authentication cookies become a temporary replacement of the user's password. For that reason, they must be properly protected during their life cycle. However, most web applications use authentication cookies with limited or no protection at all, affecting the security of the users' sessions.

Most authentication cookies are static; they do not

change during their life cycle. If an adversary captures an active authentication cookie, she can replay it to impersonate the user associated with the cookie(s) and send arbitrary requests. This problem is known as a *session hijacking* or *sidejacking* attack because the adversary takes control over an active user session. This problem is exacerbated by the fact that cookie expiration times can range from hours to weeks (e.g., "Remember Me" feature) and the lack of cookie revocation mechanisms in most web applications.

For an adversary, the easiest way to steal authentication cookies is by intercepting them while they travel across the network. Although the user's password is securely transmitted during the login process, authentication cookies are generally transmitted "in the clear" afterward. Tools such as Hamster [14] and more recently Firesheep [6] make such attacks easily automatable. While session hijacking attacks are generally conducted over insecure wireless links, they are also possible in wired networks through either direct interception of traffic in routers or potentially from network logs.

Authentication cookies can also be stolen from the user's browser. For example, an adversary can use Cross-Site Scripting (XSS) attacks [38] for this purpose. Such attacks can be prevented with the *HttpOnly* parameter, however, this parameter is not widely used [41]. In addition, Cross-Site Tracing (XST) [15] and DNS cache poisoning attacks can also be used to steal cookies from the browser even if the *HttpOnly* parameter is enabled. Finally, authentication cookies are also vulnerable to session fixation attacks, where the attacker can set a users's authentication cookie to a value the attacker controls.

The result of sidejacking attacks varies. For example, an adversary could gain access to user's private information and potentially modify it (i.e., defacement attacks). An adversary could also fabricate information (e.g., spamming and phishing attacks). Finally, an attacker could even change the user's password and completely hijack a user's account.

Many web applications are currently vulnerable to session hijacking. Table 1 lists some of the vulnerable web applications and their position in the Alexa top 100 rank. The type of web application varies from social networking to e-commerce websites.

Site	Rank
facebook.com	2
youtube.com	4
live.com	5
wikipedia.com	8
twitter.com	10
amazon.com	14
linkedin.com	21
ebay.com	22
flickr.com	36
myspace.com	62

Table 1: Alexa top 100 web applications vulnerable to session hijacking. The impact of the attack depends on the web application functionality. Note that Facebook recently announced support for site-wide HTTPS, however, it is not enabled by default (opt-in feature)

2.3 Current Solutions

The most commonly suggested solution for sidejacking attacks is the site-wide use of HTTP over a secure protocol such as TLS [11] (i.e., HTTPS). However, as Table 1 shows, many web applications do not use HTTPS for all their requests. Users could also use VPNs (Virtual Private Network) as protection against session hijacking; however, most Internet users are not familiar with setting up their own VPNs.

A small number of web applications have been taking steps to move to full HTTPS. Google [25] and more recently Facebook [26], are examples of this trend. Similarly, initiatives such as HTTP Strict Transport Security (HSTS) [17] and HTTPS Everywhere [12] also promote site-wide HTTPS support in web applications. Unfortunately, adding complete HTTPS support to existing web applications presents several challenges. First, full HTTPS support will affect the performance of the web application, as TLS is an expensive protocol. In addition, caching mechanisms do not work properly with HTTPS, further affecting performance. Additionally, browser performance may also be affected, especially when mobile platforms are considered. For example, browser caching mechanisms do not work adequately and web pages will have to be downloaded in their entirety in order to be displayed (i.e., progressive rendering does not work). Finally, network services such as network antivirus, IDSs, and content filtering [34] will also be affected.

In addition to its deployment challenges, the use of full HTTPS is not a complete solution to the session hijacking problem. HTTPS only provides confidentiality, integrity and server-side authentication[1]. Accordingly, authentication cookies are still required for client-side authentication. An adversary can still try to steal the cookies from the user's web browser. If the cookies are stolen, HTTPS will not prevent the adversary from taking control of the user's session.

Finally, confidentiality of content may not be absolutely critical for all applications. For example, the ability to see pictures on a social networking website is less critical than being able to add or delete images to an ac-

[1]Client-side authentication is possible using client certificates, but is not commonly supported on Internet environments.

count. *Accordingly, a lightweight solution guaranteeing the session integrity for webpage requests without demanding the site-wide deployment of HTTPS could be beneficial.* SessionLock [3] is an approach in this direction. SessionLock used a shared session secret to sign each request sent to the web application, thus preventing session hijacking. By using techniques such as URL fragment identifiers and URL rewriting, SessionLock offers a simple, easy to deploy solution against session hijacking without requiring full HTTPS support. However, SessionLock is not robust against active attackers and it may not work properly in some scenarios. We propose an alternative mechanism to prevent session hijacking that overcomes the limitations of SessionLock without affecting performance and deployability (see Section 5.2 for a more direct comparison). In the next section we present the details of our approach.

3 OTC Protocol

In this section, we present the One-Time Cookies (OTC) protocol. First, we describe the threat model assumed in OTC. Second, we present the protocol's design and formal definition. Third, we examine the security properties of OTC and finally, we describe implementation details.

3.1 Threat model

OTC is an HTTP session authentication protocol. It allows a web application to verify the authenticity of the requests sent by a web browser over untrusted networks. OTC specifically addresses the threat imposed by session hijacking attacks.

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 adversaries: 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, an active adversary can actively modify the requests and responses exchanged between the browser and the web application. For example, the active adversary can tamper messages, fabricate new messages and stop 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. We do not consider attacks that involve user interaction (e.g., CSRF and phishing) or attacks that exploit vulnerabilities in the supporting software (e.g., web browser, web framework, HTTP server and operating system) such as buffer overflows and malware. Also, we do not consider denial of service attacks.

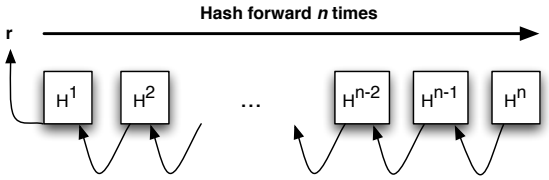

Figure 1: Hash chains are generated by hashing a secret value r forward n times. A principal Bob stores the current value of the hash chain (H^c). A participant Alice can prove knowledge of the initial secret by presenting Bob with the previous value (H^{c-1}). If Bob hashes Alice's input and generates H^c, Alice must know the initial secret due to the one-way property of hash algorithms. Bob then makes the current value H^{c-1} and waits for Alice to provide H^{c-2}.

Finally, OTC initially relies on HTTPS to initially establish its credentials. This establishment step is also currently used by many websites that rely on authentication cookies. Therefore, OTC assumes that HTTPS is established correctly and in a secure way. Our model does not consider attacks against HTTPS.

3.2 Design

OTC is designed to eliminate the vulnerabilities associated with the use of cookies as session authentication tokens. OTC offers an *intermediate* solution between the vulnerable use of authentication cookies and the secure but expensive use of site-wide HTTPS. OTC relies on *single-use authentication tokens based on modified hash chains*. Once an OTC authentication token is validated by the web application, it can not be reused for authentication, hence, preventing session hijacking attacks. Moreover, each authentication token is tied directly to the requested resource for additional security.

A hash chain [22] is a cryptographic construction used in a number of security applications and protocols. It is created by applying a cryptographic hash function $H()$ (e.g., SHA-1) multiple times to a random value r to generate a sequence of values that can be used as one-time authentication tokens (Figure 1). Hash chain security relies on the pre-image resistant (i.e., one-way) property of cryptographic hash functions.

OTC's design follows several goals. *Session integrity:* OTC should be more robust than authentication cookies and should prevent session hijacking attacks. *Performance:* OTC should have a minimal impact on the performance of the web application and the web browser.

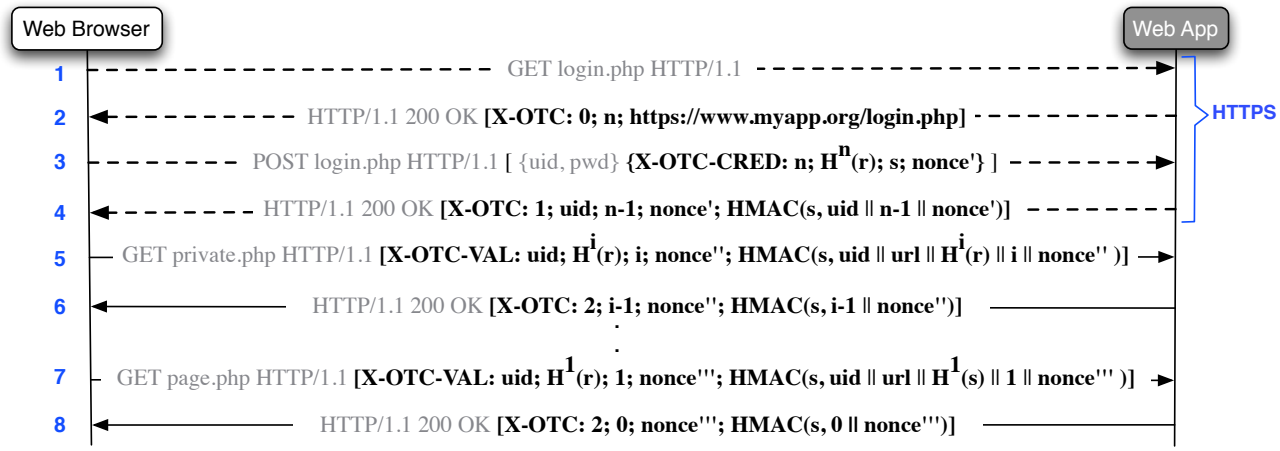

Figure 2: Flow diagram of a web session authenticated with OTC. Messages 1 to 4 show the OTC setup phase and messages 5 to 8 show the OTC authentication phase. Each HTTP request and response include an OTC header with protocol information. HMACs are used to protect the integrity of the values in the OTC headers and to tie each OTC value to a particular requested resource.

1. $C \rightarrow S : url$
2. $S \rightarrow C : [\text{X-OTC:}0, n, login\text{-}url]$
3. $C \rightarrow S : uid, pwd,$ $[\text{X-OTC-CRED:}n, H^n(r), s, nonce]$
4. $S \rightarrow C : [\text{X-OTC:}1, uid, n-1, nonce,$ $\text{HMAC}(s, uid||n-1||nonce)]$
5. $C \rightarrow S : [\text{X-OTC-VAL:}uid, H^i(r), i,$ $nonce, \text{HMAC}(s, uid||url||H^i(r)||)]$
6. $S \rightarrow C : [\text{X-OTC:}2, i-1, nonce,$ $\text{HMAC}(s, i-1||nonce)]$

C, S	:	browser, Web application
uid, pwd	:	username and password
r, n, i	:	Hash chain secret seed, length and current sequence number
s	:	shared session secret
url	:	URL of the resource requested by the client
$login\text{-}url$	:	URL of the application's login service
$H^i(x)$	:	*i-th* hash value of x, H(H(...H(x)...))
$\text{HMAC}(k, x)$	:	HMAC with key k on x
X-OTC*	:	HTTP headers used for exchanging OTC protocol information

Figure 3: *One-Time Cookie protocol:* Formal definition of the OTC protocol. OTC assumes that the setup phase (steps 1 to 4) is executed over an encrypted connection (HTTPS)

Its efficiency should be comparable to the one of authentication cookies. *Deployability:* OTC should be easy to implement in both, the web browser and the web application. It should have a low implementation cost. *Usability*: OTC should be easy to use and provide a user experience similar to the use of cookies.

The OTC protocol consists of two phases: setup and authentication. In the setup phase, the web browser generates the OTC credentials and sends them to the web application. In our design, *the browser generates the credentials to reduce the load in the web application, providing better scalability*. Both the browser and the web application store OTC state. Once the server validates the OTC credentials, the authentication phase starts. During this phase, the browser uses the OTC credentials to generate one-time cookies that are appended to each request sent to the web application. These temporary credentials are then validated by the web application to verify the authenticity of the request. This phase continues until the hash chain in the OTC credentials is completely exhausted or the user ends her session.

3.3 Formal Description

Figure 2 shows a graphical description of the OTC protocol and Figure 3 presents its formal definition. In these figures, messages 1 to 4 represent the OTC setup phase, and messages 5 to 8 represent the OTC authentication phase. The scenario depicted in these figures assumes that the web browser and the web application have not

established OTC credentials before. Therefore, if the user requests a private resource from the web application (e.g., personal profile page), the web application will redirect the user's browser to the login page for explicit authentication. During the login process, the OTC protocol will setup its credentials in both parties (setup phase). Once the user logs in successfully, OTC will be used (instead of authentication cookies) for session authentication (authentication phase).

In Step 1, the browser establishes an HTTPS session with the web application and requests the login page. HTTPS is required to provide server-side authentication and to protect the confidentiality and integrity of the user's primary authentication credentials (i.e., username and password). OTC also uses this secure connection to transmit its credentials.

In Step 2, the web application sends the login page over HTTPS. The login page includes an additional HTTP header: `X-OTC`. This header tells the browser that the web application supports the OTC protocol. This header contains three (3) values. The first value (0), is a status flag that indicates that the web application is ready to start the OTC protocol and that the browser has to generate new OTC credentials. The second value (n) corresponds to the hash chain length recommended by the web application. The third value ($login\text{-}url$) contains the URL where the browser has to send the new OTC credentials. This should be the same URL that was used by the browser to send the user's login information.

In Step 3, the browser generates new OTC credentials and sends them over HTTPS to the web application together with the user's login information (generally an HTTP POST request). The OTC credentials are sent in a new HTTP header: `X-OTC-CRED`. This header contains the following values: the hash chain length (n), the hash chain ($H^n(r)$), a session secret (s) and a nonce. The browser creates a new record in memory for the web application's domain and stores the following values: the hash chain secret seed (r), the shared session secret (s), the hash chain sequence index (i), the $login\text{-}url$ and the domain. Initially, $i = n - 1$.

In Step 4, the web application validates the user's primary authentication credentials (e.g., username and password). If this validation is successful, the web application proceeds to store in memory the new OTC credentials (s,$H^n(r)$, $i = n - 1$) together with other user's session information. Then, the web application sends a *200 OK* response to the browser including a new `X-OTC` header. This header indicates that the new OTC credentials are ready for use. This header contains a status code set to 1 (OTC credentials activated), the user id (this value is optional and application-dependent), the position of the next hash chain value expected ($n - 1$), a nonce and a HMAC (Hash-based Message Authentication Code) used to protect the integrity of the values in the header. The HMAC also provides server-side authentication by means of the session secret s. After receiving and verifying the web application response, the browser activates the OTC credentials (end of setup phase).

In Step 5, the browser requests a private resource from the web application. The request includes a new HTTP header: `X-OTC-VAL`. This header contains the user id, the next hash chain value ($H^i(r)$ where $i = n - 1$ in the first iteration), the sequence index i, a nonce, and a HMAC. In this step, the HMAC also includes the URL of the requested resource (tying the hash chain value to its specific request). After receiving the request, the web application first verifies that the sequence index included in the OTC header matches the one in memory. Upon success, the web application proceeds to verify the hash chain value. For this verification, the web application performs one hash operation over the value included in the request and compares it to the value stored in memory. If the values match, the web application continues with the verification of the HMAC using the shared secret s stored in memory. If all the previous verifications are successful, the request is considered valid. Then, the web application updates the hash chain value in memory and decrements the sequence index by one. If any of the previous verifications fails, the request is canceled and the web application redirects the browser to the login page (Step 2).

In Step 6, after a successful OTC authentication, the web application sends a 200 OK response code and the requested resource to the browser. The response also includes an `X-OTC` header with a status value set to 2 (successful OTC authentication). The header also includes the index of the next hash chain value ($i-1$) expected by the web application, a nonce and a HMAC. After verifying the header information, the browser updates its OTC state (e.g., decrements its hash chain sequence index). As shown in Figure 2 (messages 7 and 8), the browser continues using OTC for session authentication until the hash chain is exhausted ($i = 0$). Then, the web application redirects the user's browser to the login page to start a new session (Step 2). Alternately, OTC can be extended to support automatic renewal of its credentials (see Section 5.4).

3.4 Security Analysis

In this section, we discuss the security properties of the OTC protocol and examine how they address the current security threats against web session management.

The main goal of the OTC protocol is to prevent session hijacking attacks. To achieve this, OTC uses hash chains to generate single-use authentication cookies (similar to the idea of one-time password protocols).

Therefore, every request sent from a browser will include a unique authentication token that, once verified by the web application, can not be reused (the tokens become invalid immediately as opposed to the use of an expiration time like cookies). As a result, a passive adversary using tools such as Hamster or Firesheep will not be able to hijack users' sessions. Similarly, access to network logs will be of no help to adversaries.

Active adversaries could try more advanced attacks. For example, a man-in-the-middle (MITM) attack where the adversary captures the OTC value and prevents it from reaching the web application. To limit the impact of such attacks, OTC implements additional security measures. First, OTC ties each hash chain value to the request's URL by using an HMAC. The adversary can not forge the HMAC because she does not know the session secret (s). Consequently, the adversary can not use the captured OTC value for arbitrary requests, only for the original user's request [2]. Second, the browser will not release new OTC values until receiving confirmation from the web application that the current active value has been used. As before, the adversary can not forge the web application OTC confirmation header because she does not know s. Therefore, *the adversary can not impersonate the web application to the browser to obtain unused OTC values.*

From the previous paragraph, we can conclude that in order to hijack a session an adversary needs both an active hash chain value (or the hash chain secret seed r) and the shared session secret s. To formally prove that OTC does not leak either r and s during the authentication phase (Steps 5 and 6 in Figure 2), we used ProVerif [1, 2, 5], an automatic cryptographic protocol verifier. We translated a single OTC authentication transaction into Horn clauses and gave it as an input to ProVerif. The tool's output successfully confirmed that both r and s remain secret during the transaction. The Horn clause translation of the OTC transaction and the output of ProVerif are included in the Appendix of this paper.

Though the protocol guarantees the secrecy of r and s, there are other methods to obtain these values. An active adversary could try to attack the user's browser, the only place where these values are stored. Common scripting attacks such as XSS can not be used to steal OTC credentials because the OTC's browser extension does not expose these values to script languages or other browser components (e.g., stored in private objects). OTC credentials are used exclusively for authentication purposes, hence, they do not need to be shared. Moreover, session fixation attacks will not work against OTC for similar reasons. Other active attacks such as XST and DNS cache poisoning can be used to leak active OTC values; however, as we mentioned before, these values can not be used for arbitrary requests. Finally, OTC offers no protection against attacks where the adversary acquires control of the user's computer or misleads the user into sending requests to the web application (e.g., phishing and CSRF attacks).

The robustness of OTC relies on the security properties of hash chain and HMAC operations, both well-known and frequently used cryptographic constructions. There are no reported critical vulnerabilities affecting these two mechanisms. Reported vulnerabilities affecting cryptographic hash algorithms [40] do not impact hash chains because hash chains rely only in the preimage property of hash algorithms. Moreover, if the particular hash algorithm employed by OTC becomes vulnerable, OTC can be easily modified to support a different algorithm. For further protection, nonce values are added to each HMAC operation. The use of a nonce guarantees the freshness of the web application responses and makes OTC resistant to analysis and protocol attacks. As a result, we can conclude that OTC values are robust against guessing and brute force attacks and cryptanalysis.

The OTC protocol differs from traditional hash chain authentication protocols such as the Lamport's scheme [22] and S/Key [16]. Lamport's scheme and S/Key use a challenge-response approach. However, that approach is not efficient for web session authentication because it adds an extra round trip to each request. To avoid this problem, OTC maintains a synchronized state between the browser and the web application. Hence, the browser does not need a challenge from the web application to know what OTC value to send next. In addition, by using extra security checks (i.e., HMACs), OTC is more robust against attacks affecting other hash chain authentication protocols [27] (e.g., server impersonation, small "n" attacks).

Finally, OTC only provides robust client-side session authentication (via hash chains) and server-side authentication (via shared session secret). OTC does not provide integrity protection or confidentiality to the HTTP payload.

3.5 OTC implementation

As described in Section 3.2, OTC requires a client and a server component. On the server side, we implemented OTC as a plug-in for WordPress (a popular open-source web content management system). This plug-in was developed in PHP language and required approximately 300 lines of code. Most of this code, how-

[2] An adversary could still modify the payload information in the original user's request. For example, changing the text of a blog post request. To prevent this, OTC can be extended to compute a hash of the HTTP payload and include it in the HMAC to protect its integrity.

ever, was needed for integration with WordPress (OTC setup and verification required less than 100 lines of code). On the client side, we implemented OTC as an extension for Firefox (desktop and mobile versions). The Firefox extension was developed using XUL and JavaScript languages and also required approximately 300 lines of code. In addition, the Firefox extension uses the jsSHA [3] library for hash operations. Both plug-ins and their supporting documentation are currently available at `http://www.cc.gatech.edu/~idacosta/otc.html`.

We implemented OTC using a software plug-in approach for easier deployment. Installing and activating OTC support in WordPress and Firefox take only minutes. The OTC code logic on the server side is simple and it should be easy to adapt it to other web applications. OTC could also be implemented at the HTTP server level (e.g., Apache web server) to increase deployability. However, most web applications prefer to implement their own authentication mechanisms as opposed to the mechanisms offered by the web server. The reason is that the mechanisms offered by the server (e.g., Digest authentication) do not integrate well with the look and feel of the web application.

As it was mentioned earlier, the browser and the web application exchange OTC values and responses using HTTP headers. In our implementation, the `X-OTC-Val` header containing the one-time cookie looks as follows:

```
X-OTC-Val:  admin;
a6b3178501126f824bc3e268dd3f2bf33f903d3a;999;
b94d9570803d57e1ec1e4694cb3911cb0e21c551;
04bb8506b6fdff2d971bd9d4ff6a7200d45653d9
```

The corresponding `X-OTC` header containing the web application response looks as follows:

```
X-OTC: 3;998;
b94d9570803d57e1ec1e4694cb3911cb0e21c551;
42ab818029a313326eaa90c7fedf5703ba2cbe3e
```

Finally, our OTC implementation uses SHA-1 as the default cryptographic hash algorithm. However, support for other hash algorithms can be added with few modifications to our code base.

4 Experimental Evaluation

In this section we present the experimental evaluation of our implementation. Our goal is to characterize and compare the performance overheads added by OTC and current session authentication alternatives (e.g., cookies and cookies with HTTPS). First, we describe the experimental testbed used in our experiments. We then present each experiment and its results.

4.1 Experimental Testbed

Our experimental testbed consists of five servers: one web server, one proxy server and three servers for generating the test load. All the servers run Ubuntu 8.04 (Linux Kernel 2.6.24) and have 2 Quad-Core 2.00 GHz processors, 16 GB of memory and Gigabit Ethernet cards. Also, all the servers are connected to a dedicated Gigabit Ethernet switch. To simulate WAN latency values in some of our experiments, we use the Linux traffic control tool with the network emulation module (netem) and set these latencies based on RTT (Round-Trip Time) values collected experimentally from the PlanetLab network testbed.

The web server runs *WordPress 3.0* (web application). WordPress is configured with the *BuddyPress 1.2* plug-in, which adds social network features to WordPress. To support WordPress, the server uses *Apache 2.2* (HTTP server), *MySQL 5.0* (database) and *PHP 5.3* (web development language). All software in the web server use default configurations without performance optimizations. The only exception is the Apache HTTP server, which was modified to support a higher number of simultaneous requests, requests per connection and file descriptors.

The proxy server uses *Squid 3.1*, an open source web proxy and caching server. Squid is configured in a reverse proxy mode (web accelerator) to cache requests directed to the web server. The single processing core model of squid creates a performance bottleneck. To avoid this, we run three squid instances on the same server. Each instance is configured with its default parameters.

The test load is generated using *httperf 0.9* [28], a tool for measuring the performance of web servers. A total of three httperf instances (one per server) are used to generate the test load in our experiments. We wrote our own script to automate execution and data collection of the benchmarking experiments.

To evaluate our OTC implementation in the client side, we use two platforms: a laptop (MacBook Pro with a 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 2.2). In the laptop we use *Mozilla Firefox 3.6* and in the smartphone *Mozilla Firefox for mobile 4.03b*.

4.2 Experiments and Results

We evaluate three scenarios: cookies (our baseline), OTC and cookies with HTTPS. First, we present the results of our micro-benchmarks experiments in the web application, the laptop and the smartphone, focusing on mea-

[3] `http://jssha.sourceforge.net/`

Protocol	Cookies (95% c.i)	OTC (95% c. i.)
Setup time (ms)	0.194 (±0.004)	1.592 (±0.064)
Verif. time (ms)	1.110 (±0.039)	2.646 (±0.062)

Table 2: Web application (WordPress) credentials setup and verification time for cookies and OTC. Both mechanisms introduce approximately similar delay (the difference is due to WordPress memory operations used by OTC). Note: c.i. = confidence intervals.

Chain Length	100	1K	10K	100K
Laptop t (ms) (95% c.i.)	7.22 ±0.09	70.22 ±2.69	701.04 ±12.23	6957.82 ±87.01
S. phone t (ms) (95% c.i.)	49.30 ±1.70	462.43 ±11.82	4423.27 ±80.42	– –

Table 3: Time required to generate hash chains of different length in a laptop and in a smartphone. Hash chains with a length up to 10,000 (laptop) and 1,000 (smartphone) can be used without affecting the user experience.

suring the delay added by each configuration. Second, we present our macro-benchmark experiments where we focus on measuring the impact of each configuration in the performance (e.g., throughput and CPU utilization) of the web server under an increasing test load. Third, we show the results of evaluating the performance of the web server in a reverse caching scenario. Finally, each experiment was repeated multiple times to ensure the soundness of the results. Mean values and 95% confidence intervals are reported for all experiments.

4.2.1 Micro-Benchmarks

In the web application (WordPress), we first measured the time to setup the session authentication credentials. This time corresponds to the time required by the web application to have the credentials (cookies or OTC) ready for session authentication (typically after a successful user login). It includes the time to generate the cookies before sending them to the browser. For OTC, it includes the time that the OTC WordPress plug-in takes to parse and store the credentials sent by the browser. Second, we measured the verification time: the time that WordPress takes to verify cookies and OTC values for session authentication. In both cases, we used code instrumentation to measure these times.

Table 2 shows the average results and confidence intervals (20 samples per value) for setup and verification time. For the setup time, we can see that cookies require less time than OTC credentials. However, the difference is small ($\sim$ 1.4 ms) and is unlikely to impact the application's performance. Moreover, we determined that the difference is mainly due to WordPress memory operations used by the OTC plug-in. A similar situation occurs with the verification time. OTC verification took approximately 1.54 ms more than for cookies. Again, this difference is small and is mainly due to WordPress operations. These results are expected because the cookie generation and verification processes in WordPress are based on hash and HMAC operations, as in OTC. Therefore, we can conclude that the overhead added to the web application by both mechanisms is small and approximately the same.

The main difference between OTC and cookies resides in the client side. With cookies, the browser only stores the cookies and attaches them to every request. With OTC, the browser does more work: credential generation and storage, HMAC operations, and generation of hash chain values. However, these operations are fast (e.g., simple hash operations), except for the generation of the hash chain and hash chain values. Therefore, we measured the time required to generate hash chains of different lengths (100, 1,000, 10,000 and 100,000) using the OTC extension for Firefox (laptop) and Firefox for mobile (smartphone). The results are shown in Table 3. In the laptop, the OTC extension could use hash chains with a length up to 10,000 without affecting performance significantly. As expected, the hash chain computation takes additional time on the smartphone. In the smartphone, a hash chains with a length up to 1,000 can be used without noticeable delay. We note that our OTC extensions have not been optimized. For example, we could use C++ code instead of JavaScript to reduce the time to generate hash chains. Moreover, we could cache intermediate hash chain values to reduce the number of computations required (see Section 5.1).

The appropriate hash chain length depends on the web application and the user. To estimate the hash chain length required by a complex web application, we measured the number of requests with authentication cookies that are generated during Facebook sessions. For this test, we developed a simple Firefox extension and distributed it among members of our lab. The data collected (32 sessions) showed that on average, a Facebook session can generate 11.13 authenticated requests per minute (95% c.i. ±2.34) with a maximum of 28.38 requests per minute. Based on these results, we can estimate that a Facebook user may require hundreds of OTC values per session. Therefore, a hash chain with a length of 1,000 could be sufficient to maintain a Facebook session for a day.

We also ran micro-benchmark experiments (e.g., server response time, web page rendering time, CPU utilization) to measure the overhead added by HTTPS to the web application and the browsers (desktop and mobile). However, under a low test load, the difference between measured values for each configuration was not statisti-

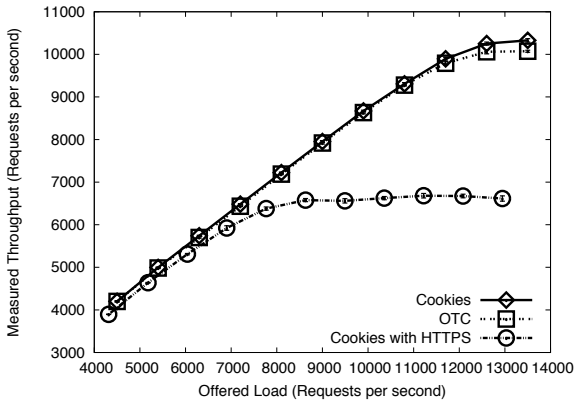

Figure 4: Request throughput supported by the web server for cookies, OTC and cookies with HTTPS configurations. While OTC and cookies have approximately the same performance, the use of HTTPS considerably reduces the throughput the web server can support.

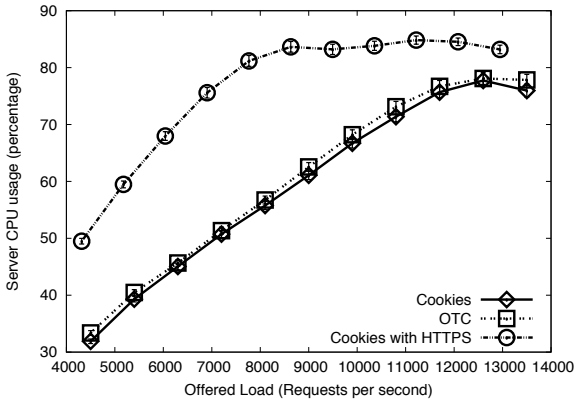

Figure 5: Web server CPU utilization for cookies, OTC and cookies with HTTPS configurations. As expected, the use of HTTPS requires more CPU time for the same load than cookies or OTC

cally significant. The impact of HTTPS was more noticeable under higher test loads, as the next section shows. Finally, storage overheads are discussed in Section 5.1.

4.2.2 Macro-benchmarks

For a more direct comparison between OTC and HTTPS, we focused on measuring performance during HTTPS steady state, avoiding the costs of HTTPS connection setup (known to be expensive [9]). 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. In addition, we used basic PHP pages with OTC and cookie support instead of using

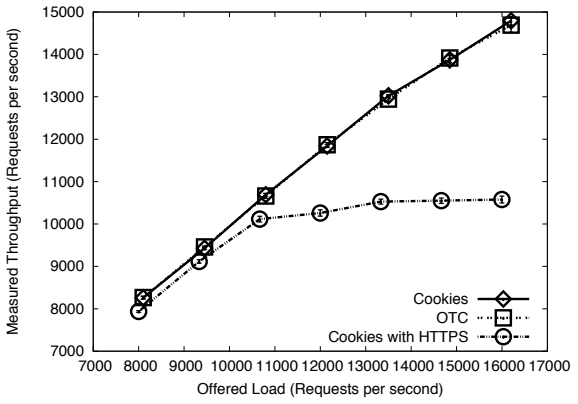

Figure 6: Request throughput supported by the web server in the presence of a reverse proxy for 3 configurations: cookies, OTC and cookies with HTTPS. While OTC and cookies benefit from the use of the reverse proxy, the performance of HTTPS quickly degrades.

WordPress pages. The reason is that our default WordPress installation was not optimized for performance (it supported less than 100 requests/sec). Finally, each experiment was repeated at least 10 times for each configuration.

The results for request throughput are shown in Figure 4. The impact of using HTTPS is considerable at high loads. When using cookies and OTC session authentication, the web application supported 10,327.72 (95% confidence interval $\pm$39.56) and 10,073.21 (95% c.i. $\pm$23.50) requests/sec respectively. However, when using cookies with HTTPS, the web application could only support approximately 6681.78 (95% c.i. $\pm$53.16) requests/sec. These results represent a 35.30 % reduction in request throughput when compared to the baseline configuration (cookies). Also, the request rate supported by OTC was only 2.46% smaller than cookies. Figure 5 shows the web server CPU utilization during the experiments for each configuration. For each configuration, the maximum CPU utilization measured was 77.74% (95% c.i. $\pm$1.04) for cookies, 78.09% (95% c.i. $\pm$1.04) for OTC and 84.84% (95% c.i. $\pm$0.72) for cookies with HTTPS. We can ee that the use of HTTPS requires more CPU time for the same throughput than HTTP. In addition, OTC and cookies require approximately the same CPU utilization during the experiments. Based on these results, we conclude that both OTC and cookies, have approximately the same performance in terms of request rate and CPU utilization. Moreover, the use of HTTPS causes a significant impact on the performance of the web server when high request loads are used.

4.2.3 Web Caching Scenario

The performance of a web application is not solely affected by the overheads introduced by HTTPS. As mentioned in Section 2.3, distributed caching and replication mechanisms do not work properly when HTTPS is used, further affecting the application's performance. We ran additional experiments to characterize how HTTPS affects the performance benefits of a reverse proxy caching web content. For these experiments, we measured the RTT between a server in California and a server in Texas using PlanetLab. We then simulated the measured latency (47.32 ms) in the network link between our WordPress server (web application) and our Squid server (reverse proxy). We also added a 5.0 ms latency between our load generator servers (clients) and the Squid server. The generated load consisted of requesting a basic PHP page with OTC and cookie support and five images.

The results for request throughput are presented in Figure 6. The reverse proxy reduces the load on the web server by caching the images for HTTP requests (cookies and OTC). In contrast, the proxy can not process requests made over HTTPS because they are encrypted and have to be forwarded to the web server. The experiments show that the server can support a maximum of 10,577.67 (95% c.i. $\pm$80.76) requests/sec for cookies with HTTPS. However, the use of the reserve proxy allows the web server to support more than 15,000 requests/sec for cookies and OTC configurations. At this point, the server CPU utilization was only approximately 17%, demonstrating that our testbed can support much higher throughput when cookies and OTC configurations are used. These results show the negative effect that enabling site-wide HTTPS could have on the performance of a web application by affecting other infrastructure components such as distributed caching for applications such as Facebook.

5 Discussion

5.1 Complexity

Compared to cookies, OTC requires additional resources. In the web application, OTC requires extra memory space to store the authentication credentials of each user. However, the space required by these credentials is small: two text strings (e.g., 160 bits) and a counter (e.g., 32 bits). This is a small requirement compared to the amount of state information that most web applications already store for their users. As our experiments showed (see Section 4.2.1), cookies and OTC add approximately the same computational overhead to the web application.

In the browser, OTC requires memory space and computation while cookies only require memory space. OTC memory requirements per domain are small (e.g., six text strings, a counter and a status flag) and similar to cookies' requirements (RFC 2695 specifies that cookies can have a size up to 4096 bytes). Regarding computation, the most expensive OTC operations in the browser are the generation of credentials and the generation of one-time cookies. The cost of these operations is directly proportional to the hash chain length. However, as our experiments showed (see Section 4.2.1), the performance overheads added by these operations is acceptable for practical hash chain lengths.

OTC can also be modified to reduce its performance overhead in the browser. For example, hash chains can be generated in advance (i.e., background process) and stored until needed. Also, the computation of hash chain values can be reduced if those values are cached during the generation of the hash chains (trading-off computation for memory space). OTC could cache hash chain values at intervals (e.g., every 100 positions) or cache all the values. By caching all the hash chain values, the overhead added by OTC in each requests will be significantly smaller and close to the overhead added by cookies. Note that caching hash chain values does not reduce OTC's security because the hash chain secret seed value (r) is already stored in the browser.

5.2 SessionLock

SessionLock [3] is another proposed technique to prevent session hijacking attacks without requiring full HTTPS support. By using a shared session secret, SessionLock signs each request sent to the web application. As OTC, the session secret is established during the user's login process over HTTPS. The session secret is then stored in the web browser by using URL fragment identifiers and URL rewriting techniques, thus avoiding the need for a browser extension.

While OTC and SessionLock have almost similar complexity in the server, SessionLock is simpler in the browser and easier to deploy (no browser extension required). However, SessionLock has several limitations. First, SessionLock does not protect against active attacks (e.g., code injection) that could compromise the session secret. OTC, in contrast, is more robust against active attacks (see Section 3.4). Second, SessionLock may not work properly with certain web applications. For example, the URL rewriting techniques will not work when binary objects (i.e., Flash) are used to generate dynamic links. Also URL rewriting could be prone to errors with complex web pages. OTC does not use URL rewriting. Third, SessionLock's shared secret could be accidentally leaked by the user or lost. In OTC, the session secrets (e.g., r and s) are only accessible to OTC func-

tions (there is no public API). The user or other browser components can not access these values. Therefore, the user can not leak these values accidentally. Finally, SessionLock has not been as thoroughly evaluated in terms of performance and compatibility with current web applications. In contrast, we have implemented OTC as a plug-in for the popular WordPress platform and as an extension for both, Firefox and Firefox for mobile. We have executed extensive performance evaluation of our OTC implementation, and our code is already available for public evaluation. Moreover, we are exploring the idea of modifying OTC to provide an OTC version that does not require a browser extension (like SessionLock).

5.3 State Synchronization In Large Web Applications

OTC requires state synchronization (i.e., sequence number and current hash chain value) between the web application and the web browser to work properly. Therefore, in large web applications, where the web application is hosted by multiple servers (e.g., server cluster), OTC state has to be synchronized among all the servers to guarantee that any server can verify the OTC values. Generally, this requirement is not difficult for servers in the same physical area. However, it is a problem for web applications with servers distributed over large geographical areas (e.g., Facebook) because of the effects of network latency. In these scenarios, it is difficult to guarantee that all the servers have the same OTC state at any given time. As a result, OTC verifications could fail, affecting the user experience.

OTC can be easily modified to solve this synchronization problem. The key idea is that the web browser will always have the latest hash chain index value (e.g., $i = 100$) and the servers will always have a similar or higher value (i.e., if the server is not synchronized, then $i > 100$). Using this idea, the OTC verification in the web application can be modified to calculate the difference between both sequence indexes. Then, the web application will perform additional hash operations during the verification process, based on the difference between the indexes. As a result, any server with the initial OTC state can verify subsequent web browser requests, even if its state is not synchronized with the web browser and the rest of the servers. A threshold of hashing operations can be established during setup to prevent small "n" attacks [19]. Finally, if the hash chain is exhausted, the web application will redirect the web browser to the login page. Once the user logs in, the new OTC credentials will be distributed to all the servers (this operation can take longer time to ensure that all the servers receive the new OTC credentials).

5.4 Automatic Renewal

The adequate hash chain length depends on the web application. For example, applications with high session activity will require longer hash chains. The web application can also recommend customized hash chain lengths based on the user's activity history and preferences. Once a hash chain is exhausted, the web application will require the user to log in again over an HTTPS connection. However, if the web application is consuming hash chains too fast and asking users to log in too often, user experience will be affected.

In scenarios where the web application can not estimate the appropriate hash chain length for its users, OTC can be modified to support automatic credential renewal without user interaction. Basically, the OTC component in the web browser will automatically generate new credentials once the hash chain has been exhausted. Then it will send it to the web application using a HTTPS connection without user intervention and using the shared session secret (s) to validate the renewal operation. In addition, to improve performance, the transmission of new OTC credentials could be executed without HTTPS. The web browser could use the shared session secret to encrypt the new credentials before sending them to the server over HTTP. We plan to explore this idea and include it in future versions of OTC.

6 Related Work

The use of cookies for client authentication and session management has raised security concerns since their adoption. In 2000, Park et al. [29] described the security threats to cookies: network, end-system and system-harvesting threats and proposed the use of secure cookies based on cryptographic techniques. A year later, Fu et al. [13] showed the problems and limitations of web client authentication mechanisms, including the risks of using Web cookies and session hijacking attacks. The authors presented several recommendations for building more robust Web client authentication schemes. However, the web industry has done very little since then to solve the problems associated with the use of Web cookies for session management. In order to raise awareness, the white-hat hacker community has released several tools that automate session hijacking attacks ("sidejacking") [6, 7, 14, 32]. Nevertheless, these problems are still present today. In a recent study of 40 Web applications, Visaggio and Blasio [39] found that few use HTTPS (15%) and that most Web applications do not protect cookies properly. As a result, most of these Web applications are vulnerable to session hijacking attacks.

A typically recommended solution against session hijacking attacks is to encrypt the HTTP payload ex-

changed between the Web browser and the Web server. While IPSec VPNs or SSH tunnels can be used for this purpose, SSL/TLS (HTTPS) is the most common alternative to encrypt HTTP traffic. However, even with HTTPS enabled, there are ways for adversaries to steal session cookies [31]. For example, several studies have shown that users tend to ignore HTTPS errors [35, 36]. Adversaries can exploit this situation to perform man-in-the-middle attacks (MITM) or entice users to establish non-HTTPS requests that will expose the session cookies. As a result, Jackson and Barth [18] proposed ForceHTTPS, a browser add-on that ensures that all session cookies are securely configured and forces all HTTPS errors to be treated as attacks, not merely configuration mistakes. Based on this work, a new web security policy mechanism, named HTTP Strict Transport Security (HSTS), is being proposed to the IETF [17]. A similar approach is used by the Electronic Frontier Foundation (EFF) tool named HTTPS Everywhere [12] that rewrites all the requests to a web site to use HTTPS. Nevertheless, MITM attacks are still possible, as demonstrated by Chen et al. [8] in their analysis of the Pretty-Bad-Proxy adversary threat. In addition, a well-known concern with the use of HTTPS for all the communication with a web server is the high impact on performance [9] and the compatibility problems with existing functionality in web applications (i.e., web caching). An alternative was recently presented by Bittau et al. [4], where the authors proposed tcpcrypt, a TCP extension designed to provide efficient, backward compatible end-to-end encryption of TCP traffic by default. While the authors demonstrated that tcpcrypt is a more robust and efficient alternative to SSL/TLS, this approach could take a long time to be tested, adopted and deployed at large scale in the Internet.

Another approach against session hijacking attacks is the design and implementation of more robust web authentication and session management protocols. Fu et al. [13] proposed a simple web client authentication scheme based on unforgeable cookies with explicit expiration time. While this scheme is secure against adaptive chosen message attacks, it is still vulnerable to replay attacks (i.e., replaying non-expired cookies) and does not provide an efficient revocation mechanism. As a result, Liu et al. [23] proposed a secure cookie protocol that offers better security guaranties than the protocol proposed by Fu. However, the authors only presented a brief security analysis of this protocol and it is not clear it provides all the security guarantees claimed (authentication, confidentiality, integrity and antireplay). In addition, this protocol is still vulnerable to server compromised. SessionLock by Adida [3] is the closest work to ours. Relying on URL fragment identifiers, time-stamps and JavaScript, SessionLock uses a session secret to sign each request to the web server. However, this mechanism requires rewriting all links in a web page, which could be computationally expensive and error-prone in the case of complex web applications (i.e., a web application that uses dynamic links generated by binary objects such as Flash). Adida also notes that SessionLock offers no protection against active attackers that could steal the session secret by injecting malicious code. Our proposed mechanism, one-time cookies, is similarly light-weight and directly addresses these shortcomings. It uses a modified hash chain construction [22] to provide better security guarantees against replay attacks. Hash chains have been used in lightweight security protocols in other domains such as sensor networks [24, 30], RFID tags [37] and more recently VoIP [10]. Our work is the first to take advantage of the security properties of hash chains in the realm of web session authentication.

7 Conclusion

Session hijacking attacks have gained a considerable amount of media attention since the release of Firesheep, creating a new push for solving this problem. As a result, several web applications have already implemented site-wide HTTPS, the recommended solution against session hijacking. While completely replacing HTTP with HTTPS will improve the overall security of the Web, it can be a challenging and complex project for some web applications. In particular, site-wide HTTPS can be difficult to justify for web applications with limited resources and low confidentiality requirements. As a result, many web applications will remain vulnerable while site-wide HTTPS is being deployed, a process that is likely to take several years. In this paper, we present One-Time Cookies, a lightweight solution to session hijacking attacks. By relying on a well-known cryptographic construction such as hash chains, OTC creates disposable authentication tokens that can not be reused, providing more robust session integrity. While OTC does not provide data confidentiality and integrity (as HTTPS does), it does provide for session integrity and prevents an adversary from replaying authentication tokens to illicitly navigate through an application. Our experimental evaluation showed that OTC is considerably more efficient than HTTPS and has approximately the same performance as current cookie-based mechanisms. In doing so, we have provided a more secure, efficient and easier to deploy alternative for managing web sessions than standard cookies.

References

[1] ABADI, M., AND BLANCHET, B. Analyzing Security Protocols with Secrecy Types and Logic Programs. In

Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (2002).

[2] ABADI, M., AND BLANCHET, B. Computer-Assisted Verification of a Protocol for Certified Email. In *Proceedings of the International Symposium on Static Analysis (SAS)* (2003).

[3] ADIDA, B. Sessionlock: Securing Web Sessions Against Eavesdropping. In *Proceedings of the ACM Conference on World Wide Web (WWW)* (2008).

[4] BITTAU, A., HAMBURG, M., AND HANDLEY, M. The Case for Ubiquitous Transport-Level Encryption. In *USENIX Security Symposium (SECURITY)* (2010).

[5] BLANCHET, B., AND CHAUDHURI, A. Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)* (2008).

[6] BUTLER, E. Firesheep. `http://codebutler.com/firesheep`, 2010.

[7] BUTLER, E., AND GALLAGHER, I. Hey Web 2.0: Start protecting user privacy instead of pretending to. ToorCon 12, `http://codebutler.github.com/firesheep/tc12`, 2010.

[8] CHEN, S., MAO, Z., WANG, Y.-M., AND ZHANG, M. Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)* (2009).

[9] COARFA, C., DRUSCHEL, P., AND WALLACH, D. S. Performance analysis of TLS Web servers. *ACM Transactions on Computer Systems (TOCS) 24*, 1 (2006).

[10] DACOSTA, I., AND TRAYNOR, P. Proxychain: Developing a Robust and Efficient Authentication Infrastructure for Carrier-Scale VoIP Networks. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2010).

[11] DIERKS, T., AND RESCORLA, E. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. http://tools.ietf.org/html/rfc5246, 2008.

[12] ELECTRONIC FRONTIER FOUNDATION. HTTPS Everywhere. `https://www.eff.org/https-everywhere`, 2010.

[13] FU, K., SIT, E., SMITH, K., AND FEAMSTER, N. Dos and Dont's of Client Authentication on the Web. *USENIX Security Symposium* (2001).

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

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

[16] HALLER, N. The S/KEY One-Time Password System. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems* (1994).

[17] HODGES, J., JACKSON, C., AND BARTH, A. HTTP Strict Transport Security (HSTS). `http://tools.ietf.org/html/draft-hodges-strict-transport-sec-02`, 2010.

[18] JACKSON, C., AND BARTH, A. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the ACM Conference on World Wide Web (WWW)* (2008).

[19] KAUFMAN, C., PERLMAN, R., AND SPECINER, M. *Network Security: Private Communication in a Public World*, second ed. Prentice Hall, 2002.

[20] KRISTOL, D., AND MONTULLI, L. RFC 2109 - HTTP State Management Mechanism. http://tools.ietf.org/html/rfc2109, 1997.

[21] KRISTOL, D., AND MONTULLI, L. RFC 2965 - HTTP State Management Mechanism, 2000.

[22] LAMPORT, L. Password authentication with insecure communication. *Communications of the ACM 24*, 11 (1981), 770–772.

[23] LIU, A., KOVACS, J., AND GOUDA, M. A secure cookie protocol. In *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)* (2005).

[24] LIU, D., AND NING, P. Multilevel μTESLA. *ACM Transactions on Embedded Computing Systems 3*, 4 (2004), 800–836.

[25] METZ, C. Google turns on SSL encryption for search. http://www.theregister.co.uk/2010/05/21/google_search_ssl_encryption/, 2010.

[26] MILLS, E. Facebook lets users turn on crypto. http://news.cnet.com/8301-27080_3-20029670-245.html, 2010.

[27] MITCHELL, C. J., AND CHEN, L. Comments on the S/KEY user authentication scheme. *ACM SIGOPS Operating Systems Review 30*, 4 (1996), 12–16.

[28] MOSBERGER, D., AND JIN, T. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review 26*, 3 (Dec. 1998), 31–37.

[29] PARK, J. S., AND SANDHU, R. Secure Cookies on the Web. *IEEE Internet Computing 4* (2000), 36–44.

[30] PERRIG, A., SZEWCZYK, R., TYGAR, J. D., WEN, V., AND CULLER, D. E. SPINS: security protocols for sensor networks. *Wireless Networks 8*, 5 (2002), 521–534.

[31] PERRY, M. Active Gmail "Sidejacking" - https is NOT ENOUGH. `http://www.securityfocus.com/archive/1/475658/30/0/threaded`, 2007.

[32] PERRY, M. 365-Day: Active Https Cookie Hijacking. DEF CON, `http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-perry.pdf`, 2008.

[33] PLANETLAB CONSORTIUM. PlanetLab — An Open Platform for Developing, Deploying, and Accessing Planetary-Scale Services. `http://www.planet-lab.org/`, 2011.

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

[35] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The Emperor's New Security Indicators. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007).

[36] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying wolf: an empirical study of SSL warning effectiveness. In *Proceedings of the Usenix Security Symposium* (2009).

[37] SYAMSUDDIN, I., DILLON, T., CHANG, E., AND HAN, S. A Survey of RFID Authentication Protocols Based on Hash-Chain Method. In *Proceedings of the International Conference on Convergence and Hybrid Information Technology* (2008).

[38] THE OPEN WEB APPLICATION SECURITY PROJECT (OWASP). Cross-site Scripting (XSS). http://www.owasp.org/index.php/Cross-site_Scripting_2010.

[39] VISAGGIO, C. Session Management Vulnerabilities in Today's Web. *IEEE Security and Privacy 8* (2010), 48–56.

[40] WANG, X. Finding collisions in the full SHA-1. In *In Proceedings of Crypto* (2005), Springer.

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

Appendix

Formal Verification using Proverif

In this section, we present the Horn Clause translation of the OTC protocol. This translation was used as input for Proverif to verify that the secrecy of the next hash value (`secretR[]`) to be used and the session key (`secretS[]`) is guaranteed. The values whose secrecy is to be verified, are given as queries to Proverif. If they are `reachable`, then secrecy is not guaranteed, while if they are `unreachable`, secrecy is guaranteed.

Horn Clause Translation of OTC

```
pred c/1 elimVar, decompData.
nounif c:x.

fun hash/1.
fun uid/1.
fun hmacR/5.
fun hmacL/2.
fun messageR/4.
fun messageL/2.

query c:secretS[].
query c:secretR[].

not c:secretS[].
not c:secretR[].

reduc

(* The attacker *)
c:x -> c:hash(x);
c:uid(x);
c:messageR(w,x,y,z);
c:messageL(x,y);
c:hmacR(v,w,x,y,z);
c:hmacL(x,y);

(* web browser *)

c:secretR[] & c:secretS[] ->
 c:messageR(uid(secretS[]), hash(secretR[]),
 url[secretS[],i], hmacR(secretS[],
 uid(secretS[]), url[secretS[],i],
 hash(secretR[]), nonce[secretS[],i]));

c:messageR(uid(secretS[]),hash(secretR[]),
 url[secretS[],i], hmacR(secretS[],
 uid(secretS[]), url[secretS[],i],
 hash(secretR[]), nonce[secretS[],i])) &
 c:messageL(x, hmacL(x, secretS[])) ->
 D:nonce[secretS[],i];


(* web server *)

c:messageR(uid(secretS[]), hash(secretR[]),
 url, hmacR(secretS[], uid(secretS[]), url,
 hash(secretR[]), nc)) -> c:messageL(nc,
 hmacL(nc, secretS[])).
```

Proverif Output

```
Completing...
ok, secrecy assumption verified:
 fact unreachable c:secretR[]
ok, secrecy assumption verified:
 fact unreachable c:secretS[]
RESULT goal unreachable: c:secretR[]
RESULT goal unreachable: c:secretS[]
```