WSKE: Web Server Key Enabled Cookies*

Chris Masone, Kwang-Hyun Baek, and Sean Smith

Department of Computer Science
Dartmouth College
Hanover, NH USA
{cmasone,jbaek,sws}@cs.dartmouth.edu

Abstract. In this paper, we present the design and prototype of a new approach to cookie management: if a server deposits a cookie only after authenticating itself via the SSL handshake, the browser will return the cookie only to a server that can authenticate itself, via SSL, to the same keypair. This approach can enable usable but secure client authentication. This approach can improve the usability of server authentication by clients. This approach is superior to the prior work on *Active Cookies* in that it defends against both DNS spoofing and IP spoofing—and does not require binding a user's interaction with a server to individual IP addresses.

1 Introduction

In this paper, we present the design and prototype of a new approach to cookie management. We developed this approach to address problems preventing currently usable Web authentication from being secure, and vice-versa.

Initially, we consider the problem of how users can authenticate themselves to servers (*user authentication*). How can end users (or their browsers) be protected from being tricked into releasing their authentication credentials to phishing Web sites? Juels et al [1] recently proposed *Active Cookies* as a solution here. This idea was based on two observations:

- Authentication based on a cookie is more usable and (potentially) more secure than authentication based on user knowledge, since the user need not remember anything and so cannot be tricked into revealing secrets to an adversary.
- In theory, the browser cannot be tricked into revealing a cookie to an adversary, since it is only supposed to send cookies back to the originating domain.

In practice, browsers can be tricked into sending cookies to a spoofed site, via DNS and IP attacks. The bulk of the *Active Cookies* work centers on addressing

* This work was supported in part by the NSF, under grant CNS-0448499, and by Sun Microsystems. The views and conclusions do not necessarily represent those of the sponsors.

this work by dispensing with DNS, and instead binding cookies and servers to specific IP addresses. It does not address the issue of IP-based attacks, such as attacks on the *Border Gateway Protocol (BGP)* used by the routers that form the backbone of the Internet to disseminate routing information. Our approach does protect against such attacks. It is important to note that cookie-based authentication schemes are already present in the wild; web sites that offer a "remember me" option at login are one such example.

In addition to user authentication, we also consider the problem of how a user can authenticate servers (*server authentication*). In theory, server-side SSL solves this problem. Server-side SSL PKI provides a flexible, scalable infrastructure for binding server identity to public keys. The SSL protocol provides a way for the user's browser to verify this binding: if a user initiates an SSL request, only the correct server should be able to complete the SSL handshake, since only the correct server should know the private key matching the public key in the presented certificate. In practice, server-side SSL does not work so well, primarily because when a server presents a certificate of questionable validity, the last line of defense is a dialog box that most users will simply click through [2]. Thus, phishers are able to spoof even SSL-protected websites with some measure of success.

In our project, we seek to address both issues by binding cookies to domain names *and public keys*. Once a user (or his browser) has accepted a server's public key, our approach applies *Key-Continuity Management (KCM)* [3] to protect any cookies set by the remote site—including cookie-based authentication credentials. Using KCM in a system means that, once a remote party is associated with a public key, steps are taken to protect the user in the event that an unexpected key is presented at a later date. By applying this methodology to server-side SSL, we reduce to one the number of times the user has to perform the SSL-certificate inspection ceremony. This, we believe, increases SSL PKI usability and helps address the server authentication problem. Perhaps more importantly, using KCM allows our approach to protect the user against IP-spoofing attacks—an improvement over Active Cookies—while also allowing DNS and SSL PKI work as intended.

In this paper, Section 2 explains our problem in more detail. Section 3 discusses our design. Section 4 presents our prototype. Section 5 presents how we evaluated it. Related work is presented in Section 6. Section 7 concludes with some ideas for future work.

2 The Problem

The Web is the primary medium today for electronic service delivery. Even services whose compromise can have serious ramifications for the parties involved—such as banking, high-value commerce, and access to health care data—now use the Web as a portal. Thus, service providers are motivated to try to assure that an alleged end user really is who she purports to be, before providing her with service. The potential value of these transactions has lead to a community of

adversaries who can find profit in subverting this authentication. Securing the process has thus become critical.

However, for a secure electronic service to make business sense, it need to attract a sufficiently large user base. If authenticating to a service is too difficult or awkward for end users, or too difficult or expensive for the deployer, then it will fail. Users will either be driven away, or driven to find some way to work around the service's security architecture [4, 5]. An unusable user authentication strategy can weaken the security of an entire system.

In theory, technologies such as client-side SSL can provide a painlesss way for users to authenticate themselves to servers, without the server learning enough to impersonate that user somewhere else. This enables a user to use one authenticator at many sites, and insulates him (somewhat) from malicious servers. In practice, however, client-side SSL requires a PKI for users at large—which appears to be practical currently only within enterprise populations (such as a corporation or a university). As a result, deployments gravitate toward knowledge-based authentication—userid and password. (However, it's not clear how "usable" passwords really are—humans are not too good at remembering such things.)

Server authentication—how users authenticate servers—is a related issue. The continued problem of Web-based phishing despite a myriad of experimental anti-phishing toolbars shows that the current technology base does not do a very good job. Ordinary users still have trouble determining if the server their browser is interacting with is in fact the bona fide representative of the service provider they intended to contact.

Besides the application-level issues (can the user figure out what the browser's UI is trying to tell them?), system designers need also worry about network-level attacks. For example, a web user typically identifies their intended destination server via a host name. The browser and the PC it's running on then use local *Domain Name Service (DNS)* resources to translate the host name to an *IP address*. The browser and its PC then use local routing resources (built up globally via the *Border Gateway Protocol (BGP)*) to determine how to send network communications to the machine with that IP address. These levels of indirection, and the global protocols that support maintenance of this distributed information, are a critical part of what makes the Internet robust and scalable. Unfortunately, these infrastructures are well-known to be vulnerable to attack. Adversaries can corrupt DNS to fool a host into contact the wrong IP address (e.g., [6, 7]). Adversaries can also corrupt BGP to fool a host into thinking that an IP address belongs to an adversary's machine (e.g., [8, 9]). Spammers are reputed to make use of BGP weaknesses in practice [10, 11]. Active Cookies, since cookies are bound to the IP address of the server that sets them, are vulnerable to these kinds of IP-based attacks. Our approach, which relies on public keys and is totally agnostic to IP addresses, is not.

Secure user authentication can enable effective server authentication. A server can echo back some user-specific personal information should she successfully authenticate. However, this breaks down if a phisher can fool a user into disclosing her authenticators.

In theory, one way to address these problems would be to use easy client-side authentication, such as passwords, over server-side SSL. This would require that, each time the user interacts with this server, she correctly interprets the browser's signals regarding whether the server has correctly carried out the handshake, whether its certificate is valid and from a trustworthy source, and whether the certificate indicates the keyholder is in fact the intended service provider. In practice, of course, this is not workable. Users cannot figure this out.

As Section 1 discusses, the recent Active Cookies work takes a different direction. When a user first establishes a channel with the server, the server deposits a cookie that embodies his authentication. However, the server binds that cookie to an IP address, not a host name. Subsequently, the browser will only disclose that cookie to a server that appears to have that IP address. Unfortunately, this approach has problems with security and usability. The approach protects against DNS attacks but is vulnerable to IP-based attacks (such as attacks on BGP). If the initial channel is to be trusted, we need a way to authenticate the server. Subsequent communications need to be encrypted, if an eavesdropper is not to learn the cookie. More critically, the approach dispenses with the flexibility, load balancing and fault tolerance enabled by DNS and multiple IP addresses; it uses IP addresses for authentication, rather than a technology such as SSL PKI that was actually designed for it. On the usability side, Active Cookies would require that web pages which use cookies have addresses with numeric IP addresses in them, as opposed to human-friendly domain names. Phishers often use such web addresses, and security professionals are trying to educate users to be suspicious of them. Requiring legitimate web applications to use numeric IP addresses would seem to be counterproductive. Thus, an ideal solution to the user authentication problem would allow DNS to do its work and map human-friendly domain names to numeric IP addresses behind the scenes.

This leaves us with the challenge: can we do better? Can we develop a usable way for users to authenticate to servers that:

- like Active Cookies, protects against phishers using Web spoofing and DNS attacks;
- unlike Active Cookies, resists BGP attacks;
- unlike Active Cookies, uses DNS, IP and server-side SSL for their intended purposes; and
- unlike passwords with server-side SSL, prevents the user from having to correctly interpret browser SSL signals each time they connect?

3 Design

To address this challenge, we propose *Web Server Key Enabled Cookies (WSKE-Cookies).* We leave DNS and IP as they are, but apply KCM to server-side SSL PKI, making server authentication easier for users. This, in turn, allows cookie-based authentication schemes—which are easier to use than passwords—to be used more safely. We are aware that WSKECookies (pronounced "Whiskey

Cookies") do not address the *registration problem*, that is the process of acquiring an authentication cookie in the first place. We consider this issue to be out of scope, and acknowledge that assuming an attacker is not privy to the initial contact between a user and a web server is accepting a risk, but it is worth noting that users of Secure Shell (SSH) have been accepting this risk for years.

The attack model against which WSKECookies defends consists of an attacker acquiring (or generating) an SSL certificate for his web server, and then using DNS or IP-spoofing attacks to route traffic destined for a target site to that server. The certificate used by the attacker will either not match the domain name to which the user is connecting and/or not be from a Certification Authority that the user's browser is configured to trust. In these cases, the browser asks the user for input about whether to drop the connection or proceed. A simple solution to the problem of protecting users' authentication cookies would be to refuse to send cookies over any connection in which errors arise during SSL session negotiation. One reason web browsers do not currently do this is compatibility; the most recent survey of SSL certificates on the web by Security Space shows that about 60% would cause warnings upon connection [12]. Our solution should not "break the web" by rendering web applications on all these servers unusable, and so we choose not to block cookies by default. Moreover, there is an attack combining DNS or IP spoofing, redirection, and cleverly crafted SSL certificates that can connect a user over SSL to a spoofed website without any warnings at all, provided that the user's initial connection goes to an insecure site [13, 14]. For instance, if the user types `www.gmail.com` into his browser, he will be connected, by default, to `http://www.gmail.com`, which can easily be DNS or IP spoofed by an attacker with no warnings. At that point, the attacker can redirect the user to an SSL site that she controls and for which she has a valid SSL certificate. If she chooses a plausible name for this site, it is likely that the user will be fooled.

Ideally, we would like to build WSKECookies as a man-in-the middle that lives at the client end, watches every https connection, remembers the domain names and public keys of servers that set cookies, and prevents cookies from being released to domains that cannot prove knowledge of the correct public key. Web browsers already ensure that cookies only go to the same domain as the one that set them, so our job then becomes to guarantee that the public key associated with a domain name does not change between visits.

The first step is to note when cookies are being set and remember the domain name and public key of the server which set them. This is not difficult given the architecture of Mozilla Firefox, our development platform. Conceptually, protecting cookies is not difficult either. Figure 2(a) outlines the architecture of the relevant portion of Firefox, and notes the ideal area where our code would hook in. After the browser has readied the outgoing https request (including the cookie), it initiates an SSL session with the server. At any point between the arrival of the server's SSL certificate in the browser and the browser's sending of the request, our code could feasibly jump in and verify that the server's key hasn't changed since the cookie was set. If the key is different, our code would

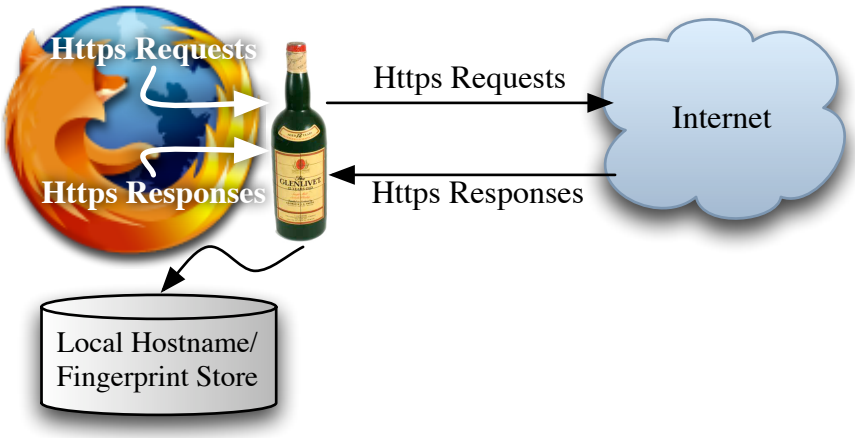

Fig. 1. WSKECookies is implemented as a Firefox extension that imposes itself between all outgoing https requests and all incoming https responses. Once a webserver has set a browser cookie via an https connection, our extension remembers the domain name and public key of that host in a local database. Every time the browser attempts to send an https request containing a cookie to a remote site, our code verifies that the current SSL connection to that site was established using the same key as the first time the user went there. If not, all cookies are removed from the request.

remove the cookie from the request and perhaps provide some feedback to the user.

The use of this framework would be similar to the Active Cookies framework. When a user initially enrolls at the server, she verifies the channel is trusted, and the server deposits a cookie that enables her authentication. The server designs their site to echo some type of personal identifier back to the user upon successful authentication. On subsequent interactions, the server regards presentation of this cookie as proof that it's that user; that user regards presentation of this information as proof that it's that server.

Unlike Active Cookies, in our framework, the user would explicitly use server-side SSL to authenticate the initial channel.

4 Prototype

Active Cookies does not require modifying the browser. Our approach does. So, we felt that it was necessary to build a proof-of-concept to demonstrate the idea. We chose the Mozilla Firefox platform (as mentioned above) because of its status as a mature but open-source framework [15, 16]. Unfortunately, the realities of the Firefox architecture provided some hurdles.

Firefox provides three notable programmer's hooks during the process of sending an https request and handling the associated response:

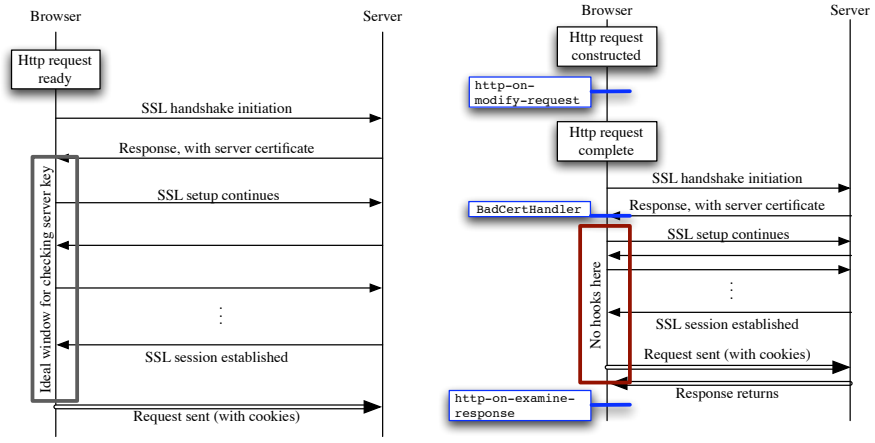

Fig. 2. (a) A ladder diagram of the relevant interactions in Firefox. The ideal period of time for our code to take effect is noted. (b) A ladder diagram of Firefox's preparation of an outgoing https request and the handling of the associates response. The three most useful programmer's hooks that are available to us during this process are also noted here.

- the `http-on-modify-request` event,
- the `http-on-examine-response` event, and
- the `BadCertHandler` object.

(See Figure 2(b)). The first two are similar to signals that can be caught and acted upon, while the last is an object that provides event handler functions which are called in response to various kinds of "bad" server certificates that may be encountered during SSL negotiation. The `BadCertHandler` object's event handling functions are not provided access to the http request, and thus cannot alter it to prevent cookies from being leaked. Thus, we are left with the two events.

The `http-on-examine-response` event fires *after* the SSL session is established (it has to be, as the request and response both had to travel over the secured channel). The remote server's certificate is therefore available in the browser. Thus, this event provides an easy way for WSKECookies to note the initial setting of a cookie by a remote server via https, and also to remember its domain name and public key for future reference. This situation is shown in Figure 3(a). The browser does not yet have any cookies set for the domain it is about to access, so no action must be taken on the outgoing request. When the response comes back, our code is notified and can cull through the response's headers for the domain name, access the server certificate used to set up the secured channel, pair this information up and store it to disk for later usage. Our implementation currently uses an XML-based flat-file database [17, 18].

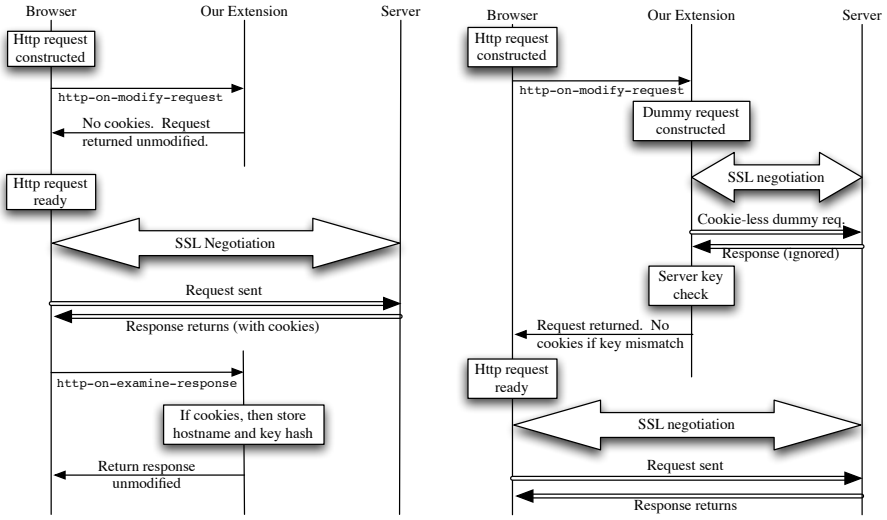

Fig. 3. (a) A ladder diagram of the browser's first interaction with a server that sets cookies via https. Since no cookies for this domain yet exist, our extension code does not need to worry about any being sent out. When a cookie is set, our extension logs the domain name and its public key fingerprint to an XML database on the local disk for later usage. (b) A ladder diagram of the browser connecting to a domain for which is has cookies it is willing to give up over https. The `http-on-modify-request` event fires, giving control to our code. WSKECookies builds a dummy request to send to the server in question, making sure no cookies are leaked. The response from this request is ignored, but the certificate is harvested and used to perform a check on the key. If it matches the key used when the cookies were set initially, our code leaves the initial request alone and allows the browser to go ahead and send it. If not, our code removes the cookies from the initial request and allows the browser to send only this modified version to the server.

As shown in Figure 2(b), the `http-on-modify-request` event fires *before* an SSL session is established with the server being accessed. This means that, at the time our code is given control, we cannot access the server's certificate, because *the browser does not know it yet*. This is not necessary behavior; rather, it is simply the order in which Firefox chooses to do things. Our current proof-of-concept works around this issue by creating a dummy request and sending that to the same URI that the original request was attempting to reach (Figure 3(b)). This dummy request has no cookies, and the response is ignored. We remember the channel object used by the dummy interaction in a hash table [19], so that we can safely ignore the proper response. The whole point is to force the browser to negotiate an SSL session with the desired server so that its certificate can be harvested. We are aware that this creates a small time-of-check-time-of-use (TOCTOU) vulnerability in our current implementation (if the attacker strikes between the time when the response to the dummy request comes back and

the sending of the original request, we will not notice), but the risk here seems negligible. In future revisions of our code, we hope to come up with a cleaner way to address this issue, probably by adding hooks into NSS (the code module that carries out the SSL handshake). In our testing, the use of these dummy requests did not affect the functionality of web applications that make use of cookies over secure channels, as long as the responses were not allowed to filter through to the browser.

5 Evaluation

5.1 Attack Resistance

To evaluate WSKECookies against possible attacks, we set up a small testbed that consists of two Apache2 Web servers and a Bind9 DNS server:

- The legitimate web server, *Bob*, holds a valid X.509 certificate that matches its domain name, `www.wske.com`.
- The attacker's web server, *Trudy*, holds a different X.509 certificate that matches the domain name, `www.wske.com`. Trudy's certificate may or may not be signed by a trusted root. We tested both cases.
- The DNS server will be used to create the effect of a DNS spoofing attack. Specifically, we can modify the DNS server so that we can direct the traffic that was meant for `www.wske.com` to either Bob or Trudy.

When Alice, a web client, connects to Bob via https, WSKECookies stores Bob's domain name, `www.wske.com`, and the fingerprint of the public key in Bob's certificate. We simulated an IP address spoofing attack by simply bringing Bob down from the network and have Trudy take over Bob's IP address and domain name. A BGP attack would have a similar effect, from Alice's point of view. Moreover, by changing the domain name `www.wske.com` to map to Trudy's IP address, we simulated a DNS spoofing attack, redirecting all the traffic intended for Bob to Trudy (see Figure 4). In both cases, WSKECookies correctly detects that the public key fingerprint in Trudy's certificate does not match Bob's public key fingerprint.

It is worth noting that cookies can also be accessed through a JavaScript interface. However, our approach is easily adapted to address this threat as well; when JavaScript code on an SSL-protected page attempts to access a protected cookie, WSKECookies could verify that the page in question was loaded from a server that presented the appropriate key.

5.2 Usability

In terms of usably protecting users' authentication cookies, WSKECookies is transparent to the user. As long as they are not being spoofed, users will see no change in their experience. If they *are* being spoofed, users will be unable to release their authentication credentials. The issue of effectively communicating

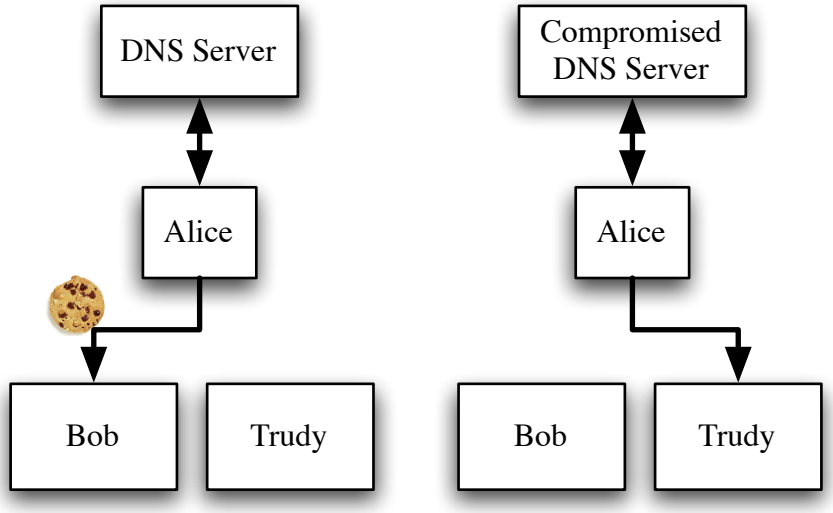

Fig. 4. Our DNS-spoofing attack scenario. (a) Our initial setup, where cookies can and should be released. (b) The "evil" setup, where DNS cannot be trusted and WSKE-Cookies protects the client.

to the user in the latter case should be explored, and will be by the final draft of this paper. Since this paper was accepted during Dartmouth's interim, subjects were unavailable for a study prior to the due date of this draft.

5.3 Deployability

In most cases, WSKECookies can be deployed transparently to the providers of web services, as no server-side changes are required. One caveat arises if a site uses a load-balancing solution with its secure web servers in which each machine has a certificate with a different public key. It is difficult for us to verify how common this situation is in the wild, but this risk exists nonetheless. We mention a possible workaround in Section 7. Another concern surrounds server certificate renewal, which commonly happens once every few years. Many websites simply purchase a new server certificate when an old one is about to expire. This certificate usually has a different public key than the old one. Thus, WSKECookies set when the old certificate was in use would all be useless when the new one comes into effect. However, given that users clear their cookies or reinstall their web browser not infrequently, web sites cannot count on long-lived authentication cookies and must structure their web applications accordingly. Thus, users may need to re-register with a web site when the certificate changes. As this happens only once every year or more, it does not seem to be that great a risk. Indeed, if a WKSECookies-like scheme came into common usage, web sites could avail themselves of certificate renewal, which allows the same key

to be rolled into the new server certificate. This is available today, though its frequency of use is unknown.

6 Related Work

Active Cookies [1] is obviously the most directly related work, and we have discussed it extensively above. There is also a plethora of work in anti-phishing, using blacklists [20], browser extensions to help users understand security indicators [21, 22], or trusted paths from the server to the user [23] to try to arm users against attackers. However, these methods all require some level of diligence and understanding on the part of the user. Our approach imposes less of a burden, and also protects against a range of DNS and IP-based spoofing attacks.

Some have suggested that websites encrypt authentication cookies with a secret key before placing them on a user's machine. In fact, this is already common practice. However, if an attacker can steal this encrypted cookie, he does not need to decrypt it all. He simply has to present it to the target website as it is, and he will be successfully authenticated. Thus, solutions such as WSKECookies are still required.

7 Conclusions and Future Work

At the end of Section 2, we laid out a challenge. The WSKECookie approach we then presented meets these criteria. Like Active Cookies, WSKECookies protects against phishers using Web spoofing and DNS attacks. Unlike Active Cookies, WSKECookies also resists BGP (and other IP-related) attacks, and uses DNS, IP and server-side SSL for their intended purposes. Unlike passwords with server-side SSL, WSKECookies prevents the user from having to correctly interpret browser SSL signals each time they connect. As such, its greater usability seems clear. WSKECookies requires no user interaction during the browsing process.

A disadvantage, of course, is that WSKECookies requires changes to the browser software. Our current proof-of-concept makes these changes via the standard extensions framework.

The work reported in this paper leaves several directions for future research.

On a basic implementation level, we want to revise our proof-of-concept code to eliminate the (admittedly small) TOCTOU vulnerability we discussed in Section 4. It would of course be possible to edit the source of Firefox and create a new binary that enables WSKECookies. However, we would prefer to keep our implementation in the form of an extension, if possible. Despite the fact that we would need to overwrite some object code in the Firefox binary at runtime to clean up our approach, we believe this should be possible [24]. A similar approach would allow us to close the JavaScript hole as well. Regardless, the fact that the existing architecture forced us into our current situation suggests some deeper philosophical questions: namely, if the SSL handshake is intended to provide the client a chance to authenticate the server, why is it that we have to hack into the SSL code to allow the client a chance to examine the certificate information

before proceeding with the request? (For that matter, the apparently standard practice of having a browser renegotiate with each https request is surprising.)

On a deeper level, we also want to empirically validate the design assumptions that underlie this framework. Can users be trusted to use the SSL signals to authenticate the server before initial enrollment? Does involving the user exactly once in the SSL ceremony make it more usable than involving the user each time? Do users really find cookie-based authentication more usable than the alternatives?

We also want to explore alternatives to our design. For example, rather than binding the cookie to the server public key, we could bind it to the server's distinguished name and the public key of the trust root; this approach would allow for more of PKI—such as revocation, renewal, and perhaps even proxy certificates—to play a role in the scheme. This approach would continue in our philosophy of, building on, rather than discarding, current network and trust infrastructure.

Code Availability

Our Firefox extension is available for download at `http://www.cs.dartmouth.edu/~pkilab/cookies/`

References

1. Juels, A., Jakobsson, M., Stamm, S.: Active cookies for browser authentication. `http://www.ravenwhite.com/files/activecookies--28_Apr_06.pdf` Visited Nov. 12, 2006. Revised version to appear in NDSS 2007.
2. Dhamija, R., Tygar, J.D., Hearst, M.: Why phishing works. In: Proceedings of SIGCHI Conference on Human Factors in Computing Systems. (2006) 581–590
3. Garfinkel, S.: Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable. PhD thesis, Massachusetts Institute of Technology (2005)
4. Good, N., Dhamija, R., Grossklags, J., Thaw, D., Aronowitz, S., Mulligan, D., Konstan, J.: Stopping spyware at the gate: a user study of notice, privacy and spyware. In: Proceedings of the Symposium on Usable Privacy and Security (SOUPS). (July 2005) 43–52
5. Yee, K.P.: Chapter 13. In: Security and Usability: Designing Secure Systems That People Can Use. O'Reilly (2005) 247–273
6. DNSSEC.NET: DNS Threats and DNS Weaknesses. `http://www.dnssec.net/dns-threats.php`
7. ISC: BIND Vulnerabilities. `http://www.isc.org/index.pl?/sw/bind/bind-security.php`
8. Murphy, S.: BGP Security Vulnerabilities Analysis. Internet Draft draft-murphy-bgp-vuln-01.txt (October 2004)
9. Nordstrom, O., Dovrolis, C.: Beware of BGP Attacks. SIGCOMM Computer Communication Review **34** (April 2004) 1–8
10. Housley, R.: Personal commmunication (April 2006)

11. Ramachandran, A., Feamster, N.: Understanding the Network-Level Behavior of Spammers. In: Proceedings of ACM SIGCOMM. (September 2006)
12. Space, S.: Secure server survey by security space and E-Soft. http://www.securityspace.com/s_survey/sdata/200608/certca.html (September 2006) Visited on Jan. 10, 2007.
13. Smith, S.W., Martini, J.C.: The Guidebook for Security Craftsmen (working title). Addison-Wesley (2007) forthcoming book material from AWL 0321434838.
14. Sirer, G. Personal Communication (2006)
15. XULPlanet.com: XULPlanet.com. http://www.xulplanet.com Visited on Nov. 12, 2006.
16. Foundation, M.: Mozilla Cross-Reference. http://lxr.mozilla.org/seamonkey/ Visited on Nov. 12, 2006.
17. Wilgus, K.: Cookie store firefox 1.5 extension. http://wigginz.com/cookiestore/ Visited on Nov. 12, 2006.
18. MonkeeSage: Basic javascript file and directory IO module v0.1. available at http://kb.mozillazine.org/Io.js Visited on Nov. 12, 2006.
19. Synovic, M.: Dev Notes : Implementing HashTable in JavaScript. http://weblogs.asp.net/ssadasivuni/archive/2003/09/17/27902.aspx Visited on Nov. 12, 2006.
20. Netcraft: Netcraft anti-phishing toolbar. http://toolbar.netcraft.com Visited Jan. 14, 2007.
21. SpoofStick: Spoofstick home. http://www.spoofstick.com/ Visited Jan. 14, 2007.
22. Close, T.: mozdev.org - petname: index. http://petname.mozdev.org Petname tool, visited Jan. 14, 2007.
23. Ye, E., Smith, S.: Trusted Paths for Browsers. In: 11th USENIX Security Symposium. (August 2002) http://www.cs.dartmouth.edu/~sws/papers/usenix02.pdf.
24. Santos, N.J.: Limited delegation (without sharing secrets) in web applications. Technical Report TR2006-574, Dartmouth College (2006)