Trusted Paths for Browsers

ZISHUANG (EILEEN) YE, SEAN SMITH, and DENISE ANTHONY
Dartmouth College

Computer security protocols usually terminate in a computer; however, the human-based services which they support usually terminate in a human. The gap between the human and the computer creates potential for security problems. We examine this gap, as it is manifested in secure Web servers. Felten et al. demonstrated the potential, in 1996, for malicious servers to impersonate honest servers. In this paper, we show how malicious servers can still do this—and can also forge the existence of an SSL session and the contents of the alleged server certificate. We then consider how to systematically *defend* against Web spoofing, by creating a *trusted path* from the browser to the human user. We present potential designs, propose a new one, prototype it in open-source Mozilla, and demonstrate its effectiveness via user studies.

Categories and Subject Descriptors: K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security*; *protection*

General Terms: Human factors, Security

Additional Key Words and Phrases: Trust path, Web browser security, HCISEC

1. INTRODUCTION

The World Wide Web has become the standard medium for delivery of electronic information services to human users. Consequently, the interface between the user and his browser plays a critical role in the effectiveness of this service delivery. User interface issues, such as flashiness, aesthetics, and ease of use, have received considerable attention. In this work, we examine the *security* aspects of Web user interfaces. Does user perception of the signals indicating a secure connection exists imply that a secure connection exists?

Preliminary versions of this paper appeared as a Dartmouth College Technical Report [Ye 2002] and as a conference paper [Ye and Smith 2002].
This work was supported in part by the Mellon Foundation, Internet2/AT&T, and by the U.S. Department of Justice, contract 2000-DT-CX-K001. However, the views and conclusions do not necessarily represent those of the sponsors.
Authors' address: Z. Ye and S. Smith, Department of Computer Science, Dartmouth College, Hanover, NH 03755; email: sws@cs.dartmouth.edu; D. Anthony, Department of Sociology, Dartmouth College, Hanover, NH 03755.

Today, the security of the Web relies on *secure socket layer (SSL)* [Rescorla 2001], a protocol that uses public-key cryptography to achieve confidentiality and integrity of messages, and (optionally) authentication of parties. In a typical "secure" Web session, the client machine authenticates the server and establishes an encrypted, MAC'd channel using SSL. After establishing the SSL channel, the Web browser displays corresponding signals to indicate that an SSL session has been set up. Typically, these signals include locking the SSL padlock, changing the protocol header to `https`, popping up warning windows, and permitting the user to call up a window containing server certificate information. The human uses these signals to make his or her trust judgment about the server.

However, if an adversarial server can provide content that causes the browser to produce similar signals without actually having an authenticated SSL session, then the user will make incorrect trust judgments. The adversary can thus subvert the secure Web session simply by creating the illusion that the browser has displayed these signals.

The effectiveness of "secure" Web sessions thus critically depends on the presence of a *trusted path* between the Web browser and its human user. Through this trusted path, the browser can communicate relevant trust signals that the human can easily distinguish from the adversary's attempts at spoof and illusion.

Background. The research that this paper reports had roots in our consideration of *public-key infrastructure (PKI)*.

In theory, public-key cryptography enables effective trust judgments on electronic communication between parties who have never met. Although the bulk of PKI work focuses on distribution of certificates, we started instead with a broader definition of "infrastructure" as "that which is necessary to achieve this vision in practice," and focused on server-side SSL PKI as perhaps the most accessible (and e-commerce critical) embodiment of PKI in our society.

Loosely speaking, the PKI in SSL establishes a trusted channel between the browser and server. Our initial set of projects [Jiang et al. 2001; Smith 2000, 2001; Smith and Safford 2001] examined how to extend the trust from the channel itself into data storage and computation at the server.

However, the "secure Web" will not work unless the *human* is able to make effective trust judgments about the server with which his or his browser is interacting. Our immediate motivation was that, for our server-hardening techniques to be effective, the human needs to determine if the server is using them; however, this issue has broader implications, as Section 6 will discuss.

Related Work. In their seminal work, Felten et al. [1997] introduced the term "Web spoofing" and showed how a malicious site could forge many of the browser user interface signals that humans use to decide the server identity. Subsequent researchers [Paoli et al. 1997] also explored this area. (In Section 2, we discuss our work in this space.)

In related work on security issues of user interfaces, Tygar and Whitten [1996] examined both the spoofing potential of hostile Java applets as well as the role of user interfaces in the security of email cryptography systems [Whitten and Tygar 1999]. Work in making cryptographic protocols more tenable to the human—including visual hashes [Perrig and Song 1999] and personal entropy [Ellison et al. 2000]—also fits into this space. In concurrent work, Yee [2002] discussed user interface design for security; Lefranc and Naccache [2003] examined using malicious Java for spoofing some SSL signals, and other groups [Fogg et al. 2002; Friedman et al. 2003; Turner 2003] have examined how users draw trust conclusions about Web sites.

The world of multilevel security has also considered issues of human-readable labels on information. The *compartmented mode workstation (CMW)* [Rome 1995] is an OS that attempts to realize this security goal (and others) within a modern windowing system. Although CMW itself is not a solution[1] for Web spoofing, the approach CMW used for labeling is a good starting point for further exploration—which we consider in Section 3.3.

Subsequent to the work reported here, our own group has been looking at security and trusted path issues regarding browsers and client-side authentication [Marchesini et al. 2003]. As this paper has gone to press, additional spoofing techniques have emerged (e.g. Secunia [2004]), as has follow-on work on spoofing defenses (e.g., Herzberg and Gbara [2004]).

This Paper. In this paper, we discuss our experience in designing, building, and evaluating trusted paths between the Web browser and the human users.

Section 2 discusses the problem that our initial investigations revealed: the current browser paradigm makes it possible for an adversarial server to subvert a user's trust judgment, by causing her browser to produce signals that are similar or identical to those produced by a secure session.

Section 3 develops criteria for systematic, effective solutions. A good solution should secure as broad as possible family of parameters that humans use to form trust judgments about servers. We discuss some solution strategies we considered and then present the one we settled on, *synchronized random dynamic (SRD)* boundaries.

Section 4 discusses how we implemented this solution in open-source Mozilla; Section 5 discusses how we validated our approaches with user studies; Section 6 offers some conclusions and discusses avenues for future work.

2. SPOOFING SERVER-SIDE SSL

In the first part of our work, we examined whether the browser technology (current when we began the experiments) permitted an adversarial server to provide material that effectively mimics the signals of a trusted, secure browser.

2.1 Spoofing Overview

To make an effective trust judgment about a server, perhaps the first thing a user might want to know is the *identity* of the server.

[1]CMW labels files according to their security levels. Since the browser would run within one security level, all of its windows would have the same label. The users still could not distinguish the server material from the browser user interface.

Can the user accurately determine the identity of the server with which his browser is interacting?

On a basic level, a malicious server can offer realistic content from a URL that disguises the server's identity. Such impersonation attacks occur in the wild: by offering spoofed material via a URL in which the spoofer's hostname is replaced with an IP address (the Hoke case [Maremont 1999; United States Securities and Exchange Commission 1999] is a good example); by *typejacking*—registering a hostname deceptively similar to a real hostname, offering malicious content there, and tricking users into connecting (the "PayPai" case [Sullivan 2000] is a good example). Furthermore, as is often pointed out [Barbalac 2000], RFC 1738 permits the hostname portion of a URL to begin with a username and password. Hoke could have made his spoof of a Bloomberg press release even more effective by prepending his IP-hostname with a "bloomberg.com" username. Many Web browser configurations will successfully parse URL `http://www.bloomberg.com@1234567/` and fetch a page from the server whose IP address, expressed as a decimal numeral, was `1234567`. (Subsequent to our research, IE suffered a bug that permitted an escape character in the URL to cause only the username to be displayed!)

However, we expected that many Web users might use more sophisticated identification techniques that would expose these attacks. Users might examine the location bar for the precise URL they are expecting; or examine the SSL icon and warning windows to determine if an authenticated SSL session is taking place; or even make full use of the server PKI by examining the server's certificate and validation information. Can a malicious server fool even these users?

Felten et al. [1997] showed that, in 1996, a malicious site could forge many of the browser's user interface signals that humans use to decide server identity, except the SSL lock icon for an SSL session. Instead, Felten et al. used a real SSL session from the attacker server to trick the user—which might expose the adversary to issues in obtaining an appropriate certificate, and might expose the hoax, depending on how the browser handles certificate validation.

Since subsequent researchers [Paoli et al. 1997] reported difficulty reproducing this work and since Web techniques and browser user interface implementation have evolved a lot since 1996, we began our work by examining whether and to what degree Web spoofing was still possible, with current technology [Ye et al. 2002].

Our experiment was more successful than we expected. To summarize our experiment, for Netscape 4.75/4.76 on Linux and Internet Explorer 5.5 on Windows 98—current when we started our work[2]—using unsigned JavaScript and DHTML, we can produce an entry link that, by mouse-over, appears to go to an arbitrary site S. If the user clicks on this link, and either his browser has JavaScript disabled or he is using a browser/OS combination that we do not

[2]Obviously, as our work becomes archived, new versions of browsers emerge. We have tested our demo in new versions: Netscape 6.0, 7.0 in Linux; and IE 6.0 in Windows 2000, Windows XP; the demo still works well. As noted, the only nit is predicting which theme (classic or modern) the user has chosen in Netscape 6.0 and 7.0, Windows XP.

support, then he really will go to site S. Otherwise, the user's browser opens a new window that appears to be a functional browser window which contains the content from site S. Buttons, bars, location information, and most browser functionality can be made to appear correctly in this window. However, the user is not visiting site S at all; he is visiting ours. The whole window is a Web page delivered by our site.

Furthermore, if the user clicks on a "secure" link from this window, we can make convincing SSL warning window appear and then displays the SSL lock icon and the expected `https` URL. Should the user click on the buttons for security information, he or she will see the expected SSL certificate information—except no SSL connection exists, and all the sensitive information that the user enters is being sent in plain text to us.

A demonstration is available at our Web site.

2.2 Technical Details

When we describe our spoofing work, listeners sometimes counter with the objection that it is impossible for the remote server to cause the browser to display a certain type of signal. The crux of our spoofing work rests in the fact that this objection is not a contradiction. For this project, we assumed that the browser has a set of proper signals it displays as a function of server properties. Rather than trying to cause the browser to break these rules, we simply use the rich graphical space the Web paradigm provides to generate harmless graphical content that, to the user, looks just like these signals.

The Princeton paper [Felten et al. 1997] seemed to imply that a malicious server could change or overwrite the client's location bar. In our initial attempts at spoofing, we tried to add our own graphical material *over* official browser signals such as the location bar and the SSL lock icon. This was not successful. We then tried to open a new window with the location bar turned off and status bar on.[3] However, this approach creates problems in modern browsers. With some Netscape Navigator configurations, turning off one bar invokes a security alarm. With Internet Explorer, we can turn off the bar without triggering alarms. But unfortunately, we cannot convincingly insert a fake location bar, since a noticeable empty space separates browser bars (where the location bar should be) and server-provided content (where our fake bar is).

Leaving *all* the bars in the new window makes spoofing impossible, and turning off *some* caused problems. However, we noticed that these problems do not occur if we turn off *all* the bars in the new window. Furthermore, when we do this, we can then replace all of them with our own fake ones—and they all appear with the correct spacing, since that's under our control. We can get fake bars simply by using images, culled from real browsers via *xv* or screen print. Since the browser happily gives its name to the server, we know whether to provide Netscape or Internet Explorer images. Our first attempts to place the images on the fake window resulted in fake-looking bars: for example, Netscape 4.75 leaves space on the right side, apparently for a scroll bar. However, creating

[3]Through subsequent email conversations, we learned that this is what the Princeton work did [Dean and Wallach 2001].

a *frame* and filling it with a background image avoids this problem. Background images get repeated; we address that problem by constraining the frame to exactly the size of the element.

We then went through a careful process of filling this window with material that looked just like the official browser elements, and correlating this display with the expected display for the session being spoofed. This work was characterized by the pattern of trying to achieve some particular effect, finding that the obvious techniques did not work, but then finding that the paradigm provided some alternate techniques that were just as effective. For one example, whenever it seemed difficult to pop up a window with a certain property, we could achieve the same effect by displaying an *image* of such a window, and using precaching to get these images to the user's machine before they are needed. This pattern made us cautious about the effectiveness of simplistic defenses that eliminate some channel of graphical display. Over and over again, we found the richness provided a workaround.

For each client platform we targeted, we carefully examined how to provide server content that, when rendered, would appear to be the expected window element. Since the user's browser kindly tells the server its OS and browser family to which it belongs, we can customize the response appropriately.

2.2.1 *Fake Interaction.* To make the fake bars convincing, we need to make them as interactive as the real ones.

We do this mainly by giving an event handler for the `onmouseover`, `onmouseout`, and `onclick` events. This is the same technique used widely to create dynamic buttons in ordinary Web pages. When the user moves the mouse over a button in our fake bars, it changes to a new look—implying that it got the focus—and we display the corresponding messages on the status bar. Similar techniques can also be applied to the fake menu bar and other places.

2.2.2 *Pop-Up Menu Bar.* If the client clicks on the fake menu bar, he will expect a pop-up menu, as in the real browser. For Internet Explorer, we construct a convincing fake pop-up menu using a `popup` object with an image of the real pop-up menu. For Netscape, we use the `layer` feature, also with an image of real pop-up menu. Genuine pop-up menus have interactive capability, as users click various options; we can use image maps to enable such interaction (although we have not implemented that yet).

Recall that to get convincing fake bars in Netscape, we needed to load them as backgrounds in frames. This technique creates a problem for spoofing pop-up menus: the fake pop-up is constrained to the frame, and the fake menu bar frame is too small for convincing menus. However, this problem has a simple solution: we replace a multiframe approach with one frame—containing a merged background image.

2.2.3 *SSL Icons and the Status Bar.* Another key to convince the client that a secure connection has been established is the lock icon on the status bar.

Our approach frees the attacker from having to get certified by a trust root and also frees him from being discovered via his certificate information. In our attack, our window displays a fake status bar of our own choosing.

Consequently, when we wish the user to think that an SSL session is underway, we simply display a fake status bar containing a lock icon.

During the course of a session, various updates and other information can appear on the status bar asynchronously (i.e. not always directly in response to a user event). When this information does not compromise our spoof, we simulate this behavior in our fake window by using the JavaScript embedded timer to check the `window.status` property and copy it into the fake status bar. For Internet Explorer, we use the `innerText` property to carry out this trick.

Netscape did not fully support `innerText` until Version 6. However, we felt that displaying some type of status information (minimally, displaying the URL associated with a link, when the mouse moves over that link) was critical for a successful spoof and (at the time of experiment) not supporting victims with browsers in the Netscape 4.7X family would overly limit our audience. So, we used the `layer` feature: the spoofed page contains layers, initially hidden, consisting of status text; when appropriate, we cause those layers to be displayed. Again, since the fake status bar is simply an image that we control, nothing prevents us from overwriting parts of it.

2.2.4 *SSL Warning Windows.*

Browsers typically pop up warning windows when users establish and exit an SSL connection. To preserve this behavior, we need to spoof warning windows.

Although JavaScript can pop up an alert window which looks similar to the SSL warning window, the SSL warning window has a different icon in order to prevent Web spoofing. To work around this in Netscape Navigator, we again use the `layer` feature. The spoofed page contains an initially hidden layer consisting of the warning window. We show that layer at the time the warning dialog should be popped up—being careful so that the raised window does not cover input fields. For Internet Explorer, we use a modal dialog with HTML that shows the same content as the warning window. Unfortunately, there is a "Web page dialog" string appended on the title bar in the spoofed window.

2.2.5 *SSL Certificate Information.*

A genuine locked SSL icon indicates that, in the current session, traffic between the browser and server is protected against eavesdropping and tampering.

To date, the most common SSL scenario is *server-side authentication*. The server possesses a key pair and a certificate from a standard trust root binding the public key to identity and other information about this server. When establishing an SSL session with a server, the browser evaluates the server certificate; if acceptable, the browser then establishes SSL session keys sharable only by the entity who knows the private key matching the public key in the certificate.

Consequently, to evaluate whether or not a "trusted" session exists, a sophisticated user will not only check for the SSL icon, but will also inspect the server certificate information: in Netscape 4.75/4.76, a user does this by clicking on the "Security" icon in the tool bar, which then pops up a "Security Info" window that displays the server certificate information; in Internet Explorer and Netscape 6, a user does this by double-clicking the SSL icon.

In our spoof, since we control the bars, we control what happens when the user clicks on icons. As a proof of concept, in our Netscape 4.75/4.76 spoof, double-clicking the security icon now opens a new window that looks just like the real security window. The primary buttons work as expected; clicking "view certificate information" causes a fake certificate window to pop up, showing certificate information of our own choosing. (One flaw in our spoof: the fake certificate window has three buttons in the upper right corner, not one.)

The same tricks would work for Internet Explorer, although we have not yet implemented that case. As future work, since Netscape's *Personal Security Manager* has been touted as a solution to client security management, it would be interesting to examine the implications of spoofing there.

2.2.6 *Location Bar Interaction.* Editable location bars are one aspect of spoofing that has been made more difficult by evolving Web technology. Here, we need to either gamble on the user's configuration, or give up on this behavior. As we noted, JavaScript cannot change a real location bar, but it can hide the real location bar, and put a fake one in the position where it is expected. The fake location bar can show the URL that the client is expecting to see. But besides displaying information, a real location bar has two common interactive behaviors: receiving input from the keyboard and displaying a pulldown history menu.

To receive input in our fake bar, we can use the standard `INPUT` tag in HTML. However, this technique creates a portability problem: how to confine the editable fake location line to the right spot in the fake location bar. The `INPUT` tag takes a `size` attribute in number of *characters*, but character size depends on the font preferences the user has selected. That is, we can use a `style` attribute to specify a known font and font size, so the input field will be the right size—but if the user has selected the option "use my fonts, not the document's," then the browser will ignore our specification and use the user's instead. We try to address this problem by deleting the location line from the location bar in our background image (to prevent giving away the spoof, in case our fake bar is smaller than the original one); by specifying reasonable fonts and font sizes; and by hoping that the users who insist on using their own fonts have not specified very small or very large ones.

More robust spoofing of editable location lines requires knowing the user's font preferences. In Netscape, this preference information is only available to signed scripts; in Internet Explorer, it is not clear if will be available at all.

Another typical location bar behavior is displaying a pulldown menu of sites the user previously visited. On Netscape 4.75/4.76, this appears to consist of places the user has recently typed into the location bar. A more complete history menu is available from the "Go" tool. Displaying a fake pulldown history menu can be done using similar technique as in the menu bar implementation, discussed above. However, we have run into one limit: users expect to be able to pull down their navigation history; however, our spoofing JavaScript does not appear to be able to access this information because our script is not signed.

One technique would be to always display a fake or truncated history. For Netscape 4.75/4.76, where the location pulldown is done via an "arrow" button to the right of the location line, we could also simply erase the arrow from our fake location bar.

2.2.7 *Covering Our Tracks.* JavaScript gives the Web designer the ability to change the default behavior of HTML hyperlinks. As noted, this is a useful feature for spoofing. By overloading the `onmouseover` method, we can show misleading status information on the status bar. By overloading the `onclick` method, a normal user click can invoke complicated computation.

We use this feature to cover our tracks, should a user have the audacity to visit our spoofed entry link with an unexpected browser/OS combination, or with JavaScript turned off.

The entry link of our experimental attack, the door that opens the attacker's world to the client, has just one line: `<a href= "realsite" onclick="return openWin()">` Bringing the mouse over this link displays "realsite" on the status line, as expected.

If the JavaScript is turned off in the user's browser, the `onclick` method will not be executed at all, so it just behaves the same as a normal hyperlink. If JavaScript is turned on, `onclick` executes our `openWin` function. This function checks the `navigator` object, which has the information about the browser, the version, the platform and so on. If the browser/OS pair is not what we want, the function returns false, and the normal execution of clicking a hyperlink is resumed, the user is brought to the real site. Otherwise the fake window will pop up.

2.2.8 *Precaching.* In order to make the browser looks real, we use true browser images grabbed from screen. Since our spoof uses a fair number of images, the spoofed page might take an unusually long time to load on a slow network. This is not desirable from an attacker's point of view. To alleviate this problem, we use precaching. Each page precaches the images needed for the next page. Consequently, when the client goes to the next page, the images necessary for spoofing are already in cache.

2.3 Further Discussion

2.3.1 *Signed Scripts.* Because it gives more control to the user, Netscape Navigator also provides an API to query about the preferences and even change preferences—but, as noted earlier, usage of these features is restricted to "signed scripts," and we did not allow that for our attack. If we relax this restriction, more interesting scenarios are possible. For example, the attacker can query the security level the browser is running and whether the image switch is turned on or not. This information can then be used to guide the attack. One can imagine scenarios where the query is embedded in an "innocent" signed page, and the information is sent back for a later attack from a different server.

Netscape 6 includes user-selectable formats; if the user does not select the "classic" format, then our spoof would be easily detectable. We have not yet figured out how to query for the user's format preference; however, given the fact that the driving force behind Web evolution has been "make the pages fun" rather than "make the pages trustable," we predict that sooner or later, querying for format will be possible. Furthermore, the majority of users never bother to change the default themes. If we target to the default themes, it would give us a pretty good chance of not being detected. Another approach here would be to exploit the fact that users probably are not clear about who controls what part of their configuration: we could simply offer a window saying "would you like classic or modern theme?" We would then provide the appropriate spoofed material.

2.3.2 *Dynamic Code Generation for HTML.* In our experimental attack, all the Web pages are statically written, and our pages are composed of frames. In some situations, it may be more convenient to only call a JavaScript function in this page, and dynamically generate HTML code for each frame. This alternative approach would offer several advantages: it would make it easier to coordinate behavior among the frames, via shared variables; some environmental variables (e.g., window size) could be queried dynamically to make the fake window fit more easily in the environment.

2.3.3 *New Technology.* After we published some of our preliminary spoofing results, two companies contacted us, claiming to offer countermeasures.

Articsoft's "WebAssurity" [ArticSoft 2002] has two parts: one signs server pages, and the other embeds in the browser and verifies signatures. This solution requires changing the server infrastructure and also would require that a malicious server cannot spoof the signals associated with a successful validation.

GeoTrust's "True Site" [GeoTrust 2003] embeds an image icon into pages from servers that register with GeoTrust. When the browser user clicks on the icon, the browser invokes a new window which establishes an SSL session to GeoTrust Web site and displays the server information. However, our techniques easily enabled a complete spoof of the GeoTrust approach—we demonstrated this for a visitor (although trademark issues and fear of DMCA prevent us from making this demonstration publicly available).

2.3.4 *Other Factors.* However, our goal was enabling users to make effective trust judgments about Web content and interaction. The above spoofing techniques focused on server *identity*. As some researchers [Ellison 1999] observe, identity is just one component for such a judgment—usually not a sufficient component, and perhaps not even a necessary component.

Arguably, issues including delegation, attributes, more complex path validation, and properties of the page source should all play a role in user trust judgment; arguably, a browser that enables effective trust judgments should handle these issues and display the appropriate material. The existence of password-protected personal certificate and key pair stores in current browsers is one example of this extended trust interface; Bugnosis [Alsaid and Marti 2002] is an entertaining example of some potential future directions.

The issue of how the human can correctly identify the trust-relevant user interface elements of the browser will only become more critical as this set of elements increases. Spoofing can attack not just perceived server identity, but *any* element of the current and future browser interfaces. In Section 6, we revisit some of these issues.

Table I. Comparison of Strategies Against Design Criteria

	Inclusiveness	Effectiveness	Minimizing User Work	Minimizing Intrusiveness
No turnoff	No	Yes	Yes	No
Backgrounds	Yes	Yes	No	No
MAC Phrase	Yes	No	No	Yes
Meta Title	No	No	Yes	Yes
Meta Window	No	No	Yes	Yes
Boundaries	No	Yes	Yes	Yes
CMW-style	Yes	No	No	Yes
SRD	Yes	Yes	Yes	Yes

3. TOWARD A SOLUTION

To address this fundamental trust problem in server-side SSL, we needed to design a more effective solution—and to see that this solution is implementable in usable technology.

Section 3.1 presents the basic framework and the need for a trusted path from browser to user. Section 3.2 presents some design criteria for this trusted path. Section 3.3 discusses some approaches we considered; Section 3.4 presents the approach we chose to prototype. Table I summarizes how these approaches measure according to these criteria.

3.1 Basic Framework

We will start with a slightly simplified model.

The browser displays graphical elements to the user. When a user requests a page P from a server A, the user's browser displays both the content of P as well as status information about the content and the channel over which P was obtained. The browser executes two functions from this input page: it displays sets of graphical elements in the window as *content* from the server; and it displays sets of graphical elements in the window as *status* about this server content. Web spoofing attacks can work because no clear difference exists between how *status* and *content* are displayed. For a page P_B from an innocent server B, a clever Web designer can offer a page P_A from a server A such that the overlap between *content*(P_A) and *status*(P_B, B) is substantial. Such overlap can trick the user into mistaking the browser's rendering of the content of P_A as status from P_B.

What makes this particularly challenging is that not only must a clear difference exist—the user must also be able to perceive this difference easily. What matters is not just the actual display of the graphical elements, but the human perception of the graphical elements. As long as the human perception of status and content have overlap, then spoofing is possible.

From the above analysis, we can see the approach to systematically stopping Web spoofing is twofold:

(1) The browser should clearly distinguish the ranges of the *content* and *status* functions, even when filtered by human perception, so that malicious collisions are not possible.

(2) The browser should make it impossible for *status* to have empty output, even when filtered by human perception, so that users can always recognize a server's attempt to forge status information.

In some sense, this is the classic *trusted path* problem. The browser software becomes a trusted computer base (TCB); and we need to establish a trusted path from the status component to the user, that cannot be impersonated by content component.

3.2 Design Criteria

We consider some criteria we believed a solution should satisfy. (Section 5.4 revisits these issues.)

First, the solution should *work*:

(1) *Inclusiveness.* We need to ensure that users can correctly recognize as large a subset of the status data as possible. Browsing is a rich experience; many parameters play into user trust judgment and, as Section 6 discusses, the current parameters may not even be sufficient. A piecemeal solution will be insufficient; we need a trusted path for as much of this data as possible.
(2) *Effectiveness.* We need to ensure that the status information is provided in a way that the user can effectively recognize and utilize it. For one example, if the status information is separated (in time or in space) from the corresponding content, then the user may already have made a trust judgment about the content before even perceiving the status data.

Second, the solution should be *low impact*:

(3) *Minimizing user work.* A solution should not require the user to participate too much. This constraint eliminates the naive cryptographic approach of having the browser digitally sign each status component, to authenticate it and bind it to the content. This constraint also eliminates the approach that users set up customized, unguessable browser themes. To do so, the users would need to know what themes are and to configure the browser for a new one instead of just taking the default one.
(4) *Minimizing intrusiveness.* The paradigm for Web browsing and interaction is fairly well established, and exploited by a large legacy body of sites and expertise. A trusted path solution should not break the wholeness of the browsing experience. We must minimize our intrusion on the content component: on how documents from servers and the browser are displayed. This constraint eliminates the simplistic solution of turning off Java and JavaScript.

3.3 Considered Approaches

Having established the problem and criteria for considering solutions, we now proceed to examine potential strategies.

3.3.1 *No Turnoff.* As discussed above, one way to defend against Web spoofing is make it impossible for an element's *status* to be empty. One possible approach is to prevent elements such as the location and status bars from being turned off in any window.

However, we rejected this approach for several reasons. First, we felt that this would be overly intrusive on Web content, and thus not be accepted by the community. Many Web sites depends on pop-up windows to display advertisements, sometimes from third parties. Requiring bars would diminish the effectiveness of the advertisements, by revealing the third parties and by decreasing the aesthetic effect. Further, suppressing "back" and "forward" buttons may be important for Web sites with specific intended browsing patterns. Second, many Web elements—such as alerts and warning windows—do not naturally have "bars" to begin with. Training users to "look for the location bar" would require them to use a different technique to authenticate these browser elements. Third, malicious servers—through tricks such as raising an image of a window—can create the illusion of a Web element with arbitrary bars, no matter what rule the browser enforces. Thus, we felt that no "turnoff" would overly constrict the display of server pages and still not cover a broad enough range of browser-user channels and malicious server techniques.

3.3.2 *Customized Content.* Another set of approaches consists of trying to clearly label the status material. One strategy here would draw from Tygar and Whitten [1996] and use user-customized backgrounds on status windows. This approach has a potential disadvantage of being too intrusive on the browser's display of server content. A less intrusive version would have the user enter an arbitrary "MAC phrase" at the start-up time of the browser. The browser could then insert this MAC phrase into each status element (e.g., the certificate window, SSL warning boxes, and so on) to authenticate it. However, this approach, being text-based, had too strong a danger of being overlooked by the user.

Overall, we decided against this whole family of approaches, because we felt that requiring the user to participate in the customization would violate the "minimal user work" constraint.

3.3.3 *Metadata Titles.* We considered having some metadata, such as page URL, displayed on the window title. Since the browser sends the title information to the machine window system, the browser can ensure that the true URL always is displayed on the window title. However, we did not really believe that users would pay attention to this title bar text; furthermore, a malicious server could still spoof such a window by offering an image of one within the regular content.

3.3.4 *Metadata Windows.* We considered having the browser create and always keep open an extra window just for metadata. The browser could label this window to authenticate it, and then use it to display information such as URL, server certificate, and so on.

Initially, we felt that this approach would not be effective, since separating the data from the content window would make it too easy for users to ignore the metadata. Furthermore, this approach would require a way to correlate the displayed metadata with the browser element in question. If the user appears to have two server windows and a local certificate window open, he or she needs to figure out to which window the metadata is referring. As we will discuss shortly, CMW uses a metadata window, and a side effect of Mozilla code structure forced us to introduce one into our design.

3.3.5 *Boundaries.* In an attempt to fix the window title scheme, we considered an alternative where we use thick color instead of tiny text. Windows containing pure status information from the browser would have a thick border with a color that indicated *trusted*; windows containing at least some server-provided content would have a thick border with another color that indicated *untrusted*. Because its content would always be rendered within an untrusted window, a malicious server would not be able to spoof status information—or so we thought. Unfortunately, this approach suffers from the same vulnerability as above: a malicious server could still offer an *image* of a nested trusted window.

3.3.6 *CMW-Style Approach.* As noted earlier, the compartmented mode workstation was a windowing OS designed to enforce multilevel security labels and information flow. CMW needs to defend against a similar spoofing problem: how to ensure that a program cannot subvert the security labeling rules by opening an image that appears to be a nested window of a different security level. To address this problem, CMW adds a separate metadata window at the *bottom* of the screen, puts color-coded boundaries on the windows and a color (not text) in the metadata window, and solves the correlation problem by having the color in the metadata window change according to the security level of the window currently in focus.

The CMW approach inspired us to try merging the boundary and metadata window scheme: we keep a separate window always open, and this window displays the color matching the security level of the window currently in focus. If the user focuses on a spoofed window, the metadata window color would not be consistent with the apparent window boundary color.

We were concerned about how this CMW-style approach would separate (in time and space) the window status component from the content component. This separation would appear to fail the effectiveness and user-work criteria: the security level information appears later and in a different part of the screen; the user must explicitly click on the window to get it to focus, and *then* confirm the status information.

What users are reputed to do when "certificate expiration" warnings pop up suggests that by the time a user clicks, it is too late.

3.4 Prototyped Approach

We liked the approaches that clearly identified the source of an element (browser, or server), since that naturally aligns with the notion of "trusted path." We also liked the colored boundary approach, since colors are more effective than text, and coloring boundaries according to trust level easily binds the boundary to the window content. The user cannot perceive the one without the other. Furthermore, each browser element—including password windows and other future elements—can be marked, and the user need not wonder which label matches which window (or which technique to use to authenticate which type of element).

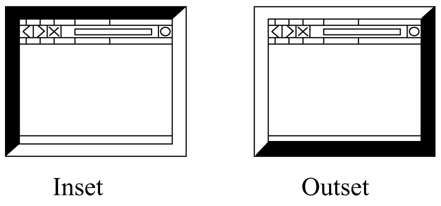

Fig. 1. *Inset* and *outset* border styles.

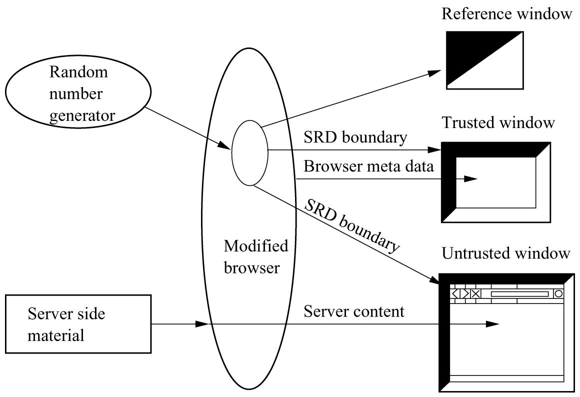

Fig. 2. The architecture of our SRD approach.

However, the colored boundary approach had a substantial disadvantage: unless the user customizes the colors in each session or actively interrogates the window (which would violate the "minimize work" criteria), the adversary can still create spoofs of nested windows of arbitrary security levels.

Consequently, we developed the *synchronized random dynamic* (SRD) boundary approach. In addition to having trusted and untrusted colors, the thick window borders would have two styles (e.g., *inset* and *outset*, as shown in Figure 1). *At random intervals* the browser would change the styles on all its windows. Figure 2 sketches this overall architecture. (One might wonder whether just randomly animating the lock icon would suffice; however, that would not protect pop-up windows and other browser elements that do not naturally have "locks.")

The SRD solution would satisfy the design criteria:

(1) *Inclusiveness.* All windows would be unambiguously labeled as to whether they contained status or content data.
(2) *Effectiveness.* Like static colored boundaries, the SRD approach shows an easy-to-recognize security label at the same time and in the same screen space as the content. Since a malicious server cannot predict the randomness, it cannot provide spoofed status that satisfies the synchronization.
(3) *Minimizing user work.* To authenticate a window, all a user would need to do is observe whether its border is changing in synchronization with the reference window.

(4) *Minimizing intrusiveness.* By changing the window boundary but not internals, the server content, as displayed, is largely unaffected.

In the SRD boundary approach, we focus on distinguishing browser-provided status from server-provided content. We build a trusted path, so that the status information presented by the browser can be correctly and effectively understood by the human user. In theory, this approach should continue to work as new forms of status information emerge.

3.4.1 *Reality Intervenes.* As one might expect, the reality of prototyping our solution required modifying this initial vision.

We first prototyped the SRD-boundary solution using Mozilla open source on Linux. We noticed that when our build of Mozilla pops up certain warning windows, all other browser threads are blocked. When the threads block, the SRD borders on all windows but the active one freeze. This freezing may generate security holes. A server might raise an image with a spoofed SRD boundary, whose lack of synchronization is not noticeable because the server also submitted some time-consuming content that slows down the main browser window so much that it appears frozen. Such windows greatly complicate the semantics of how the user decides whether to trust a window.

To address this weakness, we needed to reintroduce a metadata *reference window*, opened at browser start-up with code independent of the primary browser threads. This window is always active and contains a flashy-colored pattern that changes in synchronization with the master random bit—and with the boundaries. If a boundary does not change in synchronization with the reference window, then the boundary is forged and its color should not be trusted.

Our reference window is like the CMW-style window in that uses nontextual material to indicate security. However, ours differs in that it uses dynamic behavior to authenticate boundaries, it requires no explicit user action, and it automatically correlates to all the unblocked on-screen content.

As a side effect, having a distinguished reference window protects against a malicious server trying to confuse the user by launching many windows (real or fake, via images) with false boundaries dancing the wrong the way.

4. IMPLEMENTATION

We wanted to take our trusted path design into the real world, which requires real code, not just a sketch.

4.1 Development

Implementing our trusted path took several steps. First, we needed to add thicker-colored boundaries to all windows. Second, the boundaries needed to dynamically change. Third, the changes needed to happen in a synchronized fashion. Finally, as noted, we needed to work around the fact that Mozilla sometimes blocks browser window threads.

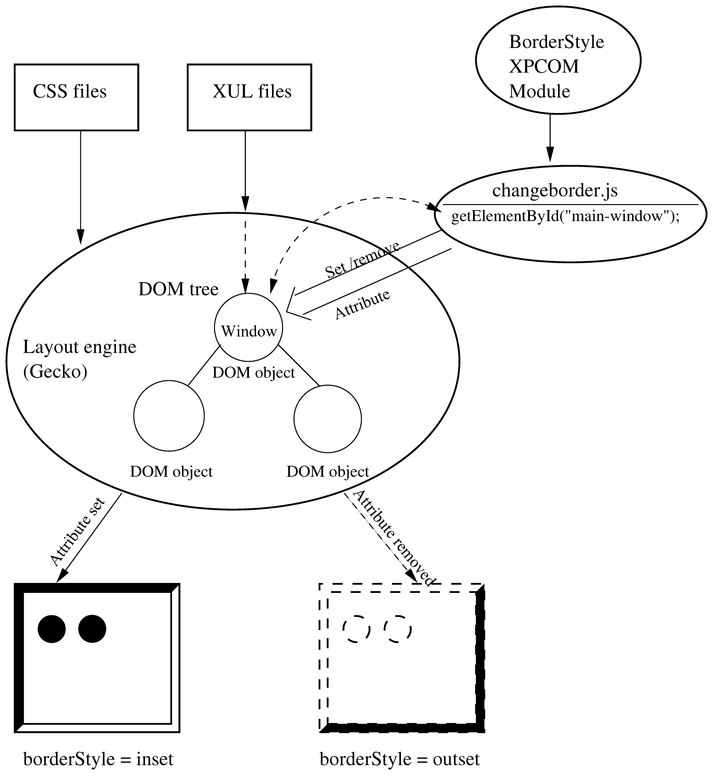

Fig. 3. This diagram shows the overall structure of our implementation of SRD in Mozilla. The Mozilla layout engine takes XUL files as input and constructs a DOM tree. The root of the tree is the window object. For each window object, JavaScript reads the random number from borderStyle module and sets or removes the window object attribute. The layout engine presents the window object differently according to the attribute. The different appearances are defined in CSS files.

Figure 3 shows the overall structure.

4.1.1 *Starting Point.* We looked at open source browsers and found two good candidates, Mozilla and Konqueror. We chose Mozilla over Konqueror for three primary reasons. First, Konqueror is not only a Web browser, but also the file manager for KDE desktop, which makes it unnecessarily complicated for our purposes. Second, Mozilla is closely related to Netscape, which has a larger market share on current desktops. Third, Konqueror only runs on Linux; Mozilla is able to adapt to several platforms. (We also considered Galeon, which is an open source Web browser using the same layout engine as Mozilla. However, when we started our experiment, Galeon was not robust enough, so we chose Mozilla instead of Galeon.)

4.1.2 *Adding Colored Boundaries.* The first step of our prototype was to add special boundaries to all browser windows. To do this, we needed to understand why browser windows look the way they do.

Mozilla has a configurable and downloadable user interface, called a *chrome*. The presence and arrangement of different elements in a window is not

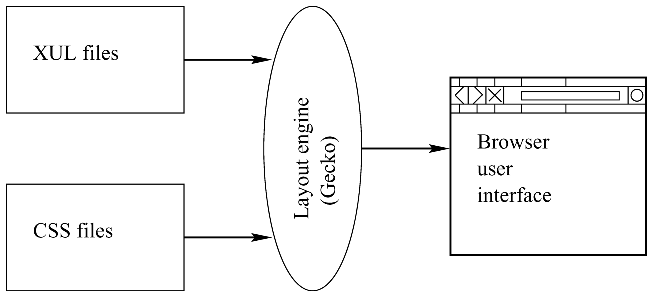

Fig. 4. The layout engine uses XUL and CSS files to generate the browser user interface.

hardwired into the application, but rather is loaded from a separate user interface description, the *XUL* files. XUL is an XML-based user interface language that defines the Mozilla user interface.

The style information—describing how a XUL document should look when prepared for viewing by a human—is included in a separate document called a *stylesheet*. Mozilla uses *cascading style sheets (CSS)* to describe what each XUL element should look like. Collectively, the set of sheets is called a *skin*. Mozilla has customizable skins. Changing the CSS files changes the look and feel of the browser. (Figure 4 sketches this structure.)

The original Mozilla only has one type of window, without any boundary. We added an *orange* boundary into the original window skin to mark the *trusted* windows containing material exclusively from the browser. Then we defined a new type of window, `external window`, with a blue boundary. We added the external window skin into the global skin file and changed the `navigator window` to invoke an `external window` instead.

As a result, all the `window` elements in XUL files will have thick orange boundaries, and all the `external windows` would have thick blue boundaries. Both the primary browsing windows as well as the windows opened by server content would be `external windows` with blue boundaries.

4.1.3 *Making the Boundaries Dynamic.* We next needed to make the boundaries change dynamically.

In the Mozilla browser, window objects can have *attributes*. These attributes can be *set* or *removed*. When the attribute is set, the window can be displayed with a different style.

To make window boundaries dynamic, we added a `borderStyle` attribute to the window. When the `borderStyle` attribute is set, the boundary style is outset; when the `borderStyle` attribute is removed, the boundary style is inset. Mozilla observes the changes in attributes and updates the display accordingly.

With a reference to a window object, browser-internal JavaScript code can automatically set the attribute and remove the attribute associated with that window. We get this reference with the method:

```
document.getElementById("windowID")
```

When browser-internal JavaScript code changes the window's attribute, the browser observer interface notices the change and schedules a browser event.

The event is executed, and the browser repaints the boundary with different style.

Each XUL file links to JavaScript files that specify what should happen in that window with each of the events in the browsing experience. We placed the attribute-changing JavaScript into a separate JavaScript file and linked it into each corresponding XUL file.

With the `setInterval` method, a JavaScript function can be called automatically at regular time intervals. We let our function be called every 0.5 s, to check a random value 0 or 1. If the random value is 0, we set the window's `borderStyle` attribute to be true; else we remove this attribute. The window's `onload` event calls this `setInterval` method to start this polling.

4.1.4 *Adding Synchronization.* All the browser-internal JavaScript files need to look at the same random number, in order to make all windows change synchronously. Since we could not get the JavaScript files in Mozilla source to communicate with each other, we used a *Cross Platform Component Object Model (XPCOM)* module to have them communicate to a single C++ object that directed the randomness.

XPCOM is a framework for writing cross-platform, modular software. As an application, XPCOM uses a set of core XPCOM libraries to selectively load and manipulate XPCOM components. XPCOM components can be written in C, C++, and JavaScript and are the basic element of Mozilla structure.

JavaScript can directly communicate to a C++ module through *XPConnect*. XPConnect is a technology which allows JavaScript objects to transparently access and manipulate XPCOM objects. It also enables JavaScript objects to present XPCOM-compliant interfaces to be called by XPCOM objects.

We maintained a singleton XPCOM module in Mozilla which tracks the current "random bit." We defined a `borderStyle` interface in *XPIDL (Cross Platform Interface Description Language)*, which only has a read-only string, which means the string only can be read by JavaScript, but cannot be set by JavaScript.

The XPIDL compiler transforms this XPIDL file into a header file and a `typelib` file. The `nsIBorderStyle` interface has a public function, `GetValue`, which can be called by Mozilla JavaScript through XPConnect. The `nsBorderStyleImp` class implements the interface, and also has two private functions, `generateRandom` and `setValue`. When a JavaScript call accesses the `borderStyle` module through `GetValue`, the module uses these private functions to update the current bit (from `/dev/random`) if it is sufficiently stale. The module then returns the current bit to the JavaScript.

Mozilla dynamically loads or unloads XPCOM modules during runtime. When a module is needed, Mozilla searches a registration category to find a pointer to this module, then create an instance of it. The registration and unregistration processes are mostly taken care of by the `Genericfactory` class. But every XPCOM module needs a factory class to glue the `Genericfactory` class with itself. `nsBorderStyleModule.cpp` is the factory class of `nsBorderStyleImp`. All the factory classes have similar structure.

In order for the JavaScript to read the random bit, we need to get a pointer to the `borderStyle` module. Our code asks for the privilege to use *XPConnect* technology, for JavaScript cannot read from C++ code directly. This privilege is automatically granted to the browser user interface code. Through XPConnect, we ask the component manager to return a pointer to the `borderStyle` module. We pass the `borderStyle` interface ID number as parameter to let the component manager know which module we need. If the instance of the `borderStyle` module has been created, the component manager simply returns a pointer to it. Otherwise, the component manager creates an instance, then returns a pointer to it.

4.1.5 *Addressing Blocking.* As noted earlier, Mozilla had scenarios where one window, such as the enter-SSL warning window, can block the others. Rather than trying to rewrite the Mozilla thread structure, we let the `borderStyle` module fork a new process, which uses the GTK+ toolkit create a reference window. When a new random number is generated, the borderStyle module passes the new random number through the pipe to the reference process. The reference window changes its image according to the random number to indicate the border style.

The idea in the GTK+ program is creating a window with a `viewport`. A `viewport` is a widget which contains two `adjustment` widgets. Changing the scale of these two `adjustment` widgets enable to allow the user only see part of the window. The `viewport` also contains a table which contains two images: one image stands for inset style, the other stands for outset. When random number is 1, we set the `adjustment` scale to show the inset image; otherwise we show the outset image.

4.2 Why This Works

To review, our solution rests on the axioms that the server material has to be displayed within a window opened by the browser, that only the browser gets to choose which type of window (trusted or untrusted) to use, and that only the browser gets to see the current random bit. A malicious server might create real or illusory inset or outset windows, but does not know what the current correct state should be. Furthermore, the existence of the distinguished reference window (Section 3.4) gives the user a distinguished reference point as to the current state.

This foundational assumption—that only the browser sees the bit—is supported by documentation. Malicious JavaScript cannot otherwise perceive the inset/outset attribute of its parent window; it only can access the attributes provided to them through the script engine implementation, and Mozilla does not provide any interface to access the window boundary attribute we added to it. The same argument holds for the Netscape-based plug-ins, since Mozilla does not provide the random bit in its API functions.

When the Mozilla layout engine Gecko reads XUL files and renders the browser user interface, it treats the `window` object as a regular XUL element, one regular *document object model* (DOM) node in the DOM tree. Browser-internal JavaScript can set or remove attributes in the `window` object. However, from the point of view of server-provided JavaScript, this `window` object is not a regular DOM element, but is rather the root object of the whole DOM tree. The root window does not provide the `window.style` interface. It also does not support any attribute functions, according to the DOM reference in Mozilla.org [2001]. Therefore, even though server-side JavaScript can get a reference of the window object and call functions like `window.open`, it cannot read or manipulate the window border style to compromise SRD boundaries.

This foundational assumption is also supported by empirical testing.

In our prototype implementation, a separate code module reads system-generated randomness and provides this string to the main browser. For additional security, we could move our random-bit module into its own address space, and thus use OS protections to help keep its bits private.

4.3 Prototype Status

Initially, we implemented SRD for the main navigator elements in modern skin Mozilla (Mozilla-0.9.2.1, current when we started) for Linux. (Subsequently, we upgraded to new Mozilla releases and Windows—which raised new issues, as discussed later.) Furthermore, we have prepared scripts to install and undo these changes in the Mozilla source tree; to reproduce our work, one would need to download the Mozilla source, run our script, then build. These scripts are available on our Web site.

4.3.1 *Inner SRD versus Outer SRD.* In the current browsing paradigm, some otherwise untrusted windows, such as the main surfing window, contain trusted elements, such as menu bars, and so on. As far as we could tell in our spoofing work, untrusted material could not overlay or replace these trusted elements, if they are present in the window.

The SRD approach thus leads to a design question: Should we just mark the outside boundaries of windows? Or should we also install SRD boundaries on individual elements, or at least on trusted ones? We use the terms *outer SRD* and *inner SRD*, respectively, to denote these two approaches.

Inner SRD raises some additional questions that may take it further away from the design criteria. For one thing, having changing, colored boundaries *within* the window arguably weaken satisfaction of the minimal intrusiveness constraint. For another thing, what about elements within a trusted window? Should we announce that any element in a region contained in a trusted SRD boundary is therefore trusted? Or would introducing such anomalies (e.g., whether a bar needs a trusted SRD boundary to be trustable depends on the boundary of its window) needlessly and perhaps dangerously complicate the user's participation?

For now, we have stayed with outer SRD. Animated GIFs giving the look and feel of browsers enhanced with outer SRD and inner SRD are available on our Web site.

4.3.2 *Upgrading for Newer Mozilla Releases.* As noted earlier, we initially implemented SRD boundaries in Mozilla 0.9.2.1, the stable release when we started our work. Later, `mozilla.org` released Mozilla 1.0, the milestone release. In order to make SRD boundaries works for the real world, we needed to upgrade for Mozilla 1.0.

Mozilla 1.0 did not change the main structure. However, small changes made the `externalwindow` tag in XUL files not recognizable. Instead, we used an `external` attribute; when a window is set with the `external` attribute, the boundary color is blue, means the window content is from the server, so it is untrustworthy.

The newer Mozilla fixed a structural problem that led to a bug in SRD for the older Mozilla. In the older Mozilla, `alert`, windows, `confirm` windows, and `prompt` windows are all handled by the same code, regardless of whether the server page content or the browser invokes them—which made it hard for our code to determine whether to label each as "trusted" or "untrusted." However, Mozilla 1.0 added a *security manager* component, which we use to keeps track who is the caller of the `alert` window. If it is the server content, we mark the `alert` window as "untrusted," otherwise, we mark it as "trusted."

In hindsight, the difficulty in upgrading our modifications is not surprising: if systems are not designed for security properties, adding them afterward is generally difficult.

4.3.3 *Porting to Windows.* If our SRD code is going to have an impact in the real world, it also needs to work on Windows. So, we also ported our SRD modifications to Mozilla on Windows.

Mozilla on Windows uses *Cygwin* to compile and run the Linux code for the Windows platform. In order to make SRD boundaries work in Windows, we needed to solve two problems. First, we needed a good random number generator, to replace `/dev/random`. Second, the reference window process in Linux used the GTK+ graphic toolkit, which needed to be reimplemented using Windows process.

To replace `/dev/random` in Windows, we used EGADS [Secure Software, Inc.], which contains an implementation of a pseudorandom number generator and an entropy gateway. EGADS has two parts: an entropy-collecting server and a client. The EGADS server is installed as a Windows system service; when the client calls `egads_init`, the server is launched and starts to collect entropy. The client side can read the entropy through an anonymous pipe. EGADS comes as a *dynamic link library* (DLL), compiled with Visual C++. A special tag should be added into `Makefile.win`, so that Mozilla is aware that EGADS is a Windows DLL.

The reference window process is implemented in Visual C++, containing several classes. The reference window process needs read the random bit out of the `borderStyle` XPCOM module. Since the Mozilla/Windows build process did not support an anonymous pipe, we used a named pipe and specified that only one process can read that named pipe. As soon as the pipe is created, the reference window process launches and links to that pipe.

The actual program has two threads, sharing one integer. One thread takes care of reading the random bit and update the sharing integer, the other thread takes care of reading the sharing integer and updating the images. The reason we need two threads is the pipereading thread is blocked whenever the pipe is empty.

4.4 Known Issues

Our current prototype has several areas that require further work. We present them in order of decreasing importance.

(As discussed earlier, we have already fixed the most critical known issue—correctly labeling Alert windows—identified in our preliminary reports.)

4.4.1 *Signed JavaScript.* Above, we discussed how server-provided JavaScript cannot access the random bit through the `window.style` attribute, because the script engine does not implement this interface. However, *signed* JavaScript from the server can ask for privileges to use XPConnect, which can enable JavaScript to directly connect to the browser's internal modules without using the script engine. The user can then choose to grant this privilege or not. If the user grants the privilege, then the signed JavaScript can access the `borderStyle` module and read the random bit.

To exploit this, an attacker would have to open an empty window (see below) or simulate one with images, and then change the apparent boundary according to the bit. For now, the user can defend against this attack by not granting such privileges; however, a better long-term solution is simply to disable the ability of signed JavaScript to request this privilege.

4.4.2 *Chrome Feature.* Mozilla added a new feature *chrome* to the `window.open` method. If a server uses the JavaScript then Mozilla will open an empty window without any boundary. The `chrome` feature lets the server eliminate the browser default chrome and thus take control of the whole window appearance. However, this new window will not be able to synchronize itself with the reference window and the other windows. Furthermore, in our experiments, this new window could not respond to the right mouse click and other reserved keystrokes, like *Alt+C* for copy under Linux. So far, the chromeless window is not a threat to SRD boundaries.

However, Mozilla is living code. The Mozilla developers work hard to improve its functionality; and the behavior of the chrome feature may evolve in the future[4] in ways that are bad for our purposes.

Consequently, a future issue we plan for our code is to ensure that our internal browser code puts SRD boundaries on all windows, even chromeless ones.

4.4.3 *Pseudosynchronization.* Another consequence of real implementation was imprecise synchronization. Within the code base for our prototype, it was not feasible to coordinate all the SRD boundaries to change at precisely the same real-time instant. Instead, the changes all happen within an approximately 1-s interval. This imprecision is because only one thread can access the XPCOM module; all other threads are blocked until it returns. Since the

[4]For example, the XUL vulnerability announced as we went to press merits further exploration [Secunia 2004].

JavaScript calls access the random value sequentially, the boundaries change sequentially as well.

However, we actually feel this increases the usability: the staggered changes make it easier for the user to perceive that changes are occurring.

4.4.4 *Other Attack Avenues.* We note that in this study, we assume that the adversary accesses the user's machine only via responses to the user's Web requests; we also assume that services and security protections provided by this interface work as advertised. If the adversary finds new flaws in JavaScript protections, or buffer overflow problems on how the browser handles certain inputs, or finds some other avenue to inject code or trigger malicious activity on the client's platform, then she may very well be able to subvert browser protections such as our solution (e.g., by using a hostile executable and an OS hole to spy on the random bit process, and report back to the server site).

Protecting against such software and OS security issues is outside the scope of this paper, although exploring how to graft a browser-level trusted path onto an OS-provided trusted path, for scenarios where the OS can reasonably protect against mutually hostile applications, is an interesting area of future work.

Examining the interval time is another area of future work. For example, with an interval of 0.5 s, if a user only looks at a spoofed page for 5 s, then a malicious server's guess at dancing might be correct for 1 in 2^{10} users.

5. USABILITY

The existence of a trusted path from browser to user does not guarantee that users will understand what this path tells them. Consequently, we conducted user studies to evaluate the usability of communicating security information via the SRD boundary.

Because our goal is to effectively defend against Web spoofing, our group plans future tests that are not limited to the SRD boundary approach, but would cover the general process of how humans make trust judgments, in order to provide more information on how to design a better way to communicate security-related information.

5.1 Test Design

The design of the SRD boundary includes two parameters: the *boundary color* and the *synchronization*. The parameters express different information: the boundary color indicates the source of the material; the synchronization indicates whether the source is trustworthy, that is, whether it is the expected/reliable source for this material.

In our tests, we vary the two parameters to determine whether the users can understand the information each parameter tries to express. We vary the boundary color between two states: *orange* implies the source is the browser (and thus trusted to report status information); *blue* implies the source is the server (and thus not trusted).

We vary the synchronization parameter between three states. If the border changes *synchronously*, then the material is from the expected source (and thus is trustworthy). If the border changes *asynchronously*, then the material is not from the expected source (and thus is not trustworthy). If the border is *static*, then the material is not from the expected source (and thus is not trustworthy).

According to our semantics, a trusted status window should have two signals: an *orange* boundary color and *synchronized* changes. Combining the variations for each of the two parameters (two colors and three change patterns) leads to six possible combinations. We eliminate the cases where *neither* of the two signals of a trusted window is present (blue asynchronous and blue static), leaving us to test four parameter combinations: (1) orange-static boundary; (2) orange-synchronized boundary; (3) blue-synchronized boundary; (4) orange-asynchronous boundary. Only the second case indicates a trusted window because it has both of the signals indicating a trustworthy and trusted source.

We tested the four different signal combinations under two different methods for checking the security of a Web site: (1) viewing the certificate window; and (2) observing the pop-up warning window (i.e., the window that warns users they are entering an SSL session). Users were told to initiate an SSL session by going to a Web site from which they would typically access their email.

We wanted to test whether users could correctly interpret the SRD signals in the two different methods of checking security information, and also to determine whether the reference window was redundant for the SRD approach to work.

We then ran two four-part test sessions. In the first test session, we turned off the reference window and used only the SRD boundary in the main surfing window as a synchronization reference. Users observed the SRD boundary in the certificate window, alternating (in different instances) among the four different signal combinations described above. In the second test session, we examined the full SRD approach and left the reference window on as a synchronization reference. Users observed the SRD boundary in the security-warning window, alternating among the four different signal combinations.

We also simulated the CMW-style approach. In the conventional CMW approach, the mouse has to be clicked on the window to get it to focus. In our test, we used *mouse-over*, which gets the information to the user sooner. In this case, the status information provided by the reference window arrives at the same time when the user move the mouse into the window. Boundaries were static; however, a reference window always showed the color indicating the source of the window to which the mouse points. Using the same color signals as above, orange indicates a trusted source (the browser); blue indicates an untrusted source. When users point the mouse to the pop-up warning window or the certificate window, the reference window will be orange if the information is coming directly from the browser.

5.2 User Study

We conducted a total of nine sessions (four signal combinations times two security methods, plus one CMW-style test) with a total of 37 undergraduate and graduate student volunteers. Each subject participated in all of the test sessions. Subjects first answered a brief questionnaire regarding demographic characteristics, as well as questions about their typical computer use and general knowledge of computer security. Next, subjects were given a brief introduction to security information (e.g., what a certificate is and what it does during an SSL session). Then they were introduced to the SRD boundary approach and informed of the two parameters (orange color and synchronized boundary changes) that would signal a trusted window. Subjects also viewed the original Mozilla user interface, in order to become familiar with the buttons and window appearance. The information on basic security issues, the SRD boundary approach and the Mozilla interface, was provided to subjects in a three-page handout, while the experiment leader talked through the information. This introductory information session lasted approximately 5–10 min in which subjects could also ask clarification questions.

The scenario for the test sessions was for users to access the Web server from which they would access their email account. Before checking email, they must verify that the browser is indeed communicating with the secure email server. Users never actually accessed their email or entered any private information during the test sessions. Instead, they started our modified browser and entered an SSL session with the server. As described above, in the first test, subjects checked the server certificate that the browser appeared to present. The certificate window popped-up with each of the four different boundary combinations. In the second test, subjects observed a pop-up reference window (opened with the browser) and the pop-up security-warning window (opened when entering an SSL session), which demonstrated each of the four different boundary combinations. In the third test, subjects observed a pop-up reference window for any color change when they put their mouse over the pop-up security-warning window.

Users were asked to observe the windows for 5 s before they answered a series of questions regarding what they observed of the two parameters in the window boundaries and whether they thought the window was secure (from the trusted source). The entire test took approximately 1 h. Subjects volunteered after responding to advertisements about the study in campus buildings. Subjects were paid $10 for their participation.

User Characteristics. The 37 subjects included both undergraduate ($n = 21$) and graduate ($n = 16$) students and ranged in age from 18 to 40, with a mean age of 23. Over half of the subjects were computer science (CS) or engineering majors ($n = 22$), though these subjects performed no differently (based on statistical comparisons) than subjects from other major areas of study. Subjects were asked to self-rate their own expertise as Internet users on a scale of 1 = expert, 2 = knowledgeable, 3 = inexperienced. 27% rated themselves as an expert and 73% as knowledgeable. (In future work, we plan to add more granularity here, in case users were tending to underrate their expertise.) However, there were no statistically significant differences in results across different types of users.

The subjects are all students on a campus with ubiquitous wired and wireless networks. Students are required to own computers, and they demonstrate a higher-than-average computer and Web usage than the average American computer user. So, while the subjects of our study are somewhat more advanced

Table II. Percent of Subjects Identifying Trusted and Untrusted Sources of Security Information Across Three Methods

% of Subjects Who	Accurately Identified Trusted Window	Accurately Identified Untrusted Window
Method 1: SRD boundary in certificate and main browser window	70%	87%
Method 2: SRD Boundary in pop-up reference and security-warning windows	89%	89%
Method 3: CMW-style color change in reference window with mouse-over of warning window	81%	No test

than the average user, our sample is fairly representative of the campus on which the study was done. Since our subjects have higher-than-average knowledge, this user study is a conservative test of the SRD method and is somewhat analogous to an experimental study in medicine of high-risk groups. In such studies of high-risk groups, only those subjects who have the greatest chance to benefit from an experimental drug, for example, are included in the study. If the drug produces no obvious benefits even among high-risk subjects, then there is no need to conduct the study within lower risk groups.

Some of our questions dealt with the subjects' computer use and security knowledge. 92% of subjects spend at least 1 h per day on the Web, in addition to email. 97% have purchased products online, and nearly all have done so in the last 6 months. 65% have heard the phrase SSL (all of the experts and half of the knowledgeable users; CS majors more likely than nonmajors to have heard of SSL). 50% of those who have heard of SSL say they know what SSL means.

When submitting private information online, only 25% say they *always* check security features on their browser. 36% *sometimes* check security features; 39% *rarely or never* check the security features.

Of those who sometimes or always check security features, 51% check the URL for `https`; 70% check the *lock icon*; 35% check the *certificate*.

SRD Test Results. Across the three test sessions, subjects accurately identified the signal of a trusted window 80% of the time and accurately identified an untrusted window 87% of the time.

Table II summarizes the results for each method. Subjects were most successful in accurately identifying a trusted window in method two in which the SRD boundary is displayed in both the pop-up reference window and the pop-up security-warning window. Subjects in method two also accurately identified untrusted windows.

In paired-samples t-tests, subjects were significantly more accurate in method two than method one ($t = 2.2$, $df = 35$, $p < 0.05$). While accuracy was also greater in method two compared to method three, this difference is not statistically significant.

Test Analysis. In the SRD tests, subjects observed the signals under two different conditions: between the certificate and main browser windows; between the reference window and the warning window. In both cases they had to assess "trustworthiness" of the windows based on the SRD boundary's color changes and synchronization. In both cases, across the different methods for checking security, the users were consistently accurate.

We had expected that the reference window might be too distracting for users. To our surprise, users were significantly more accurate using the reference window—and almost two-thirds of the subjects ranked method two over method one for ease of use. Since the SRD signals were the same, why did using the reference window achieve more accuracy? Some users commented that the reference window made the two SRD signals more observable and easier to evaluate. Further testing would be interesting.

Although method three achieved good accuracy as well, about half of the users thought that during the real browsing process, the security information brought by this approach might be easier to ignore, because method three required user action in moving the mouse over the pop-up warning window. In particular, significant gaps exist in time, space, and user action between when and where the user perceives a window's content, and when and where they receive its status information (later, on a different part of the screen, after mouse focus).

Research in HCI has explored the impact of on-screen separation of data in terms of human eyesight [Dix et al. 1997]. Humans have two kinds of light sensitive photoreceptors, *rods* and *cones*. Since rods dominate peripheral vision but are unable to resolve fine details, our ability to read or distinguish falls off inversely as the distance from our point of focus. This loss of discrimination sets limits on the amount that can be seen without moving the eyes. A user concentrating on the middle of the screen cannot be expected to read helpful information displayed elsewhere on the screen. In a regular browser session, a user's eyes tend to focus on the center of the main browser window. The CMW method requires users to frequently move their eyes from the center of concentration to the reference window during the quick pop-up-click process, which might not be as effective to deliver security information.

Further research here—as well as tests designed to measure the effectiveness of security signals when the user is browsing quickly and thinking about something else—would also be interesting.

5.3 User Study Conclusions

Overall, the usability results support the *minimal user work* property of our SRD approach: it is easy to learn even for people unfamiliar with security issues. The SRD does not require users to do much work; all they need to do is observe. The status information reaches them automatically. No Web browser configuration or details are involved. The primary and most difficult task for increasing computer security remains how to get users to understand why it is important for them to check for security signals in the first place.

5.3.1 *User Learn Quickly.* Other valuable feedback from our user study was that all users with Web experience and regular Internet use learned quickly about evaluating security signals. After reading the brief instructions, the vast majority understood the meaning of the signals they would evaluate during the test sessions. Others said they fully understood the signals after the first few test sessions.

Our subjects were very interested to learn about what they could do to detect and verify the security of private browser sessions. Subjects who were not familiar with terms like SSL or even with the standard signals of the lock icon or `https` told us they were pleased to learn about these security features, and planned to use them in the future. Computer scientists typically conclude that the reasons users do not use security features is because (1) they do not care and (2) security features are too hard to learn to use (e.g., PKI tools [Tygar and Whitten 1996]). While these reasons may be true, our findings suggest that at least part of the explanation for general users' lack of use of security features is that they are unaware of what types of security features already exist (see also Friedman et al. [2003]) or how to interpret accurately what the security features are telling them.

5.3.2 *Reference Is Better.* Most of our users felt it was better to have the reference window, because it made the synchronization parameter easy to observe. The reference window opens earlier than the main window, so it attracts users' attention. This result is somewhat ironic when one considers that we only added the reference window because it was easier than rewriting Mozilla's thread code.

5.3.3 *Dynamic Is Better.* The dynamic effect of SRD boundary increases its usability. The human users pay more attention to the dynamic items in Web pages, which is why many Web sites use dynamic techniques. In our user study, some people did not even notice that a static window boundary existed in the first session test.

5.3.4 *Automatic Is Better.* The user comments on CMW-style approach simulation also indicates that delivering security information without requiring user action could be more reliable in real-world browser sessions. Recent research further supports this idea since there is some mismatch between what users say they do when browsing, and what they actually do. For example, many users base evaluations of credibility on superficial aspects of Web sites, such as visual design, layout, and text size [Fogg et al. 2002; Turner 2003]. Delivering security signals via automatic visual displays, such as in the SRD display, appears to match the type of information users already pay attention to and evaluate.

5.4 Further Exploration of the Solution Space

Now that we have demonstrated the existence of a trusted path problem with server-side SSL, and prototyped and validated one solution, it would be fruitful to explore the solution space more thoroughly.

One area for further examination is our *design criteria* (Section 3.2). For example, we asserted that restrictions such as turning off JavaScript or requiring *all* windows to have status and location bars would be too intrusive on Web content—but what would a study of users or "in-the-wild" Web sites show?

Another area for further examination is alternate solution strategies. Our SRD approach introduced a way for users to easily authenticate browser elements by using data which is unpredictable from the server side. Changing the style boundary is just one instantiation of this idea. Other approaches include less intrusive random animation in the border (although less intrusive may mean less noticeable), or choosing the whole browser "theme" or "skin" at random (although we suspect that this may annoy users, and also not necessarily be noticeable if a window or element is from the wrong theme). Another angle would be to indicate status of an element via an easily perceived item that cannot be changed by the server: in recent personal communication, K. Yee has suggested changing the cursor symbol.

Our current user studies examined whether or not users "got it." In practice, users often understand security, but rank other things (such as aesthetics or ease of use) ahead of it. Subsequent studies that measure whether users still understand the security message when they are focused on a stressful deadline, or whether users will leave SRD (or other security features) enabled when given the choice to disable them, would be interesting.

6. CONCLUSIONS AND FUTURE WORK

The security of the standard "secure" Web paradigm critically depends on a trusted path between the browser and the user. Our work has shown that this path can be easily spoofed by a malicious server, and has offered one validated, prototyped solution to this problem.

A systematic, effective defense against Web spoofing requires establishing a trusted path from the browser to its user, so that the user can conclusively distinguish between genuine status messages from the browser itself and maliciously crafted content from the server.

Such a solution must effectively secure all channels of information the human may use as parameters for his or her trust decision; must be effective in enabling user trust judgment; must minimize work by the user and intrusiveness in how server material is rendered, and be deployable within popular browser platforms.

Any solution which uses static markup to separate server material from browser status (and does not otherwise constrain the channels) cannot resist the image spoofing attack. In order to prove the genuineness of browser status, the markup strategy has to be unpredictable by the server. Since we did not want to require active user participation, our SRD solution obtains this unpredictability from randomness.

This, we believe our SRD solution meets these criteria. We offer this work back to the community, in hopes that it may drive more thinking and also withstand further attempts at spoofing and phishing, which (as this paper goes to press) is increasingly becoming a critical topic (e.g., Lininger and Vines [2005]).

This research also suggests many new avenues of research.

Browser design. The difficulty of carrying out and maintaining our prototyped solution (both general Web technology as well as specific browser codebases are moving targets) supports the oft-repeated wisdom that security needs to be built in from the beginning, rather than added in afterwards. A rethinking of browser design and structure, to make trusted paths and other security functionality easier to build and more likely to remain secure against future technology evolution would be an interesting line of inquiry.

Parameters for trust judgment. The existence of a trusted path from browser to user does not guarantee that the browser will tell the user true and useful things.

What is reported in the trusted path must accurately match the nature of the session. Unfortunately, the history of the Web offers many scenarios where issues arose because the reality of a browsing session did not match the user's mental model. Invariably this happens because the deployed technology is a richer and more ambiguous space than anyone realizes. For example, it is natural to think of a session as "SSL with server *A*" or "non-SSL". It is interesting to construct "unnatural" Web pages with a variety of combinations of framesets, servers, 1×1-pixel images, and SSL elements, and then observe what various browsers report. For one example, on Netscape platforms we tested, when an SSL page from server *A* embedded an image with an SSL reference from server *B*, the browser happily established sessions with both servers—but only reported server *A*'s certificate in "Security Information." Subsequently, it was reported [Bonisteel 2001] that many IE platforms actually use different validation rules on some instances of these multiple SSL channels. Another issue is whether the existence of an SSL session can enable the user to trust that the data *sent back to the server* will be SSL protected [Norman 2002].

What is reported in the trusted path should also provide what the user needs to know to make a trust decision. For one example [Ellison 2000], the Palm computing "secure" Web site is protected by an SSL certificate registered to Modus Media. Is Modus Media authorized to act for Palm computing? Perhaps the server certificate structure displayed via the trusted path should include some way to indicate delegation. For another example, the existence of technology (or even businesses) that add higher assurance to Web servers (such as our WebALPS [Jiang et al. 2001; Smith 2000, 2001] work) suggests that a user might want to know properties in addition to server identity. Perhaps the trusted path should also handle attribute certificates.

Other uncertain issues pertaining to effective trust judgment include how browsers handle certificate revocation [Weiser 2001] and how they handle CA certificates with deliberately misleading names [Norman 2002].

Access control on UI. Research into creating a trusted path from browser to user is necessary, in part, because Web security work has focused on what machines know and do, and not on what humans know and do. It is now unthinkable for server content to find a way to read sensitive client-side data, such as their system password; however, it appears straightforward for server content to create the illusion of a genuine browser window asking for the user's password. Integrating security properties into document markup is an area of ongoing work; it would be interesting to look at this area from a spoof-defense point of view.

Multilevel security. It is fashionable for younger scientists to reject the Orange Book [Department of Defense 1985] and its associated body of work regarding multilevel security as being archaic and irrelevant to the modern computing world. However, our defense against Web-spoofing is essentially a form of MLS: we are marking screen elements with security levels and trying to build a user interface that clearly communicates these levels. (Of course, we are also trying to retro-fit this into a large legacy system.) It would be interesting to explore this vein further.

Visual hashes. In personal communication, Perrig suggests using visual hash information [Perrig and Song 1999] in combination with various techniques, such as metadata and user customization. Hash visualization uses a hash function transforming a complex string into an image. Since image recognition is easier than string memorization for human users, visual hashes can help bridge the security gap between the client and server machines, and the human user. We plan to examine this in future work.

Digital signatures. Another interesting research area is the application of spoofing techniques to digital signature verification tools. In related work [Kain et al. 2002], we have been examining how to preserve signature validity but still fool humans. However, both for Web-based tools, as well as non-Web tools that are content rich, spoofing techniques might create the illusion that a document's signature has been verified, by producing the appropriate icons and behavior. Countermeasures may be required here as well.

Formal model of browser content security. Section 3.1 discussed the basic framework of distinguishing browser-provided content from server-provided content rendered by the browser. However, formally distinguishing these categories raises additional issues, since much browser-provided content still depends on server-provided parameters. More work here could be interesting.

ACKNOWLEDGMENTS

We are grateful to Yougu Yuan, for his contribution in the Web spoofing experiment; Robert Camilleri, for his help with the user studies; James Rome, for his advice on CMW; and Drew Dean, Carl Ellison, Ed Feustel, Steve Hanna, Terry Hayes, Eric Norman, Adrian Perrig, Eric Renault, Jesse Ruderman, Bill Stearns, Doug Tygar, Mark Vilardo, Dan Wallach, Russell Weiser and the anonymous referees, for their helpful suggestions.

REFERENCES

ALSAID, A. AND MARTI, D. 2002. Detecting web bugs with bugnosis: Privacy advocacy through education. In *Proceedings of the 2nd Workshop on Privacy Enhancing Technologies*, San Fransicsco, CA. Springer-Verlag, Berlin.

ARTICSOFT LIMITED. 2000 WebAssurity. Online resource. `http://www.articsoft.com/webassurity.htm`.

BARBALAC, R. 2000. Making something look hacked when it isn't. *The Risks Digest 21*, 16 (Dec.).

BONISTEEL, S. 2001. Microsoft browser slips up on SSL certificates. Online resource. `http://www.computeruser.com/news/01/12/27/news4.html`.

DEAN, D. AND WALLACH, D. 2001. Personal communication.

DEPARTMENT OF DEFENSE. 1985. *Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD.

DIX, A., FINLAY, J., ABOWD, G., AND BEALE, R. 1997. *Human-Computer Interaction*, 2 ed. Prentice Hall, Englewood Cliffs, NJ.

ELLISON, C. 1999. The nature of a usable PKI. *Computer Networks 31*.

ELLISON, C. 2000. Personal communication.

ELLISON, C., HALL, C., MILBERT, R., AND SCHNEIER, B. 2000. Protecting secret keys with personal entropy. *Future Generation Computer Systems 16*.

FELTEN, E., BALFANZ, D., DEAN, D., AND WALLACH, D. 1997. Web spoofing: An internet con game. In *The 20th National Information Systems Security Conference*, Baltimore, MD.

FOGG, B., SOOHOO, C., DANIELSON, D., MARABLE, L., STANFORD, J., AND TAUBER, E. 2002. *How do People Evaluate a Web Site's Credibility? Results from a Large Study*. Tech. Rep., Consumer WebWatch/Stanford Persuasive Technology Lab.

FRIEDMAN, B., HURLEY, D., HOWE, D., FELTEN, E., AND NISSENBAUM, H. 2003. User's conceptions of web security: A comparative study. In *ACM/CHI2002 Conference on Human Factors and Computing Systems*, Minneapolis, MN. Extended abstracts.

GEOTRUST, INC. 2003. True site: Identity assurance for Web sites. Online resource. `http://www.geotrust.com/true_site/index.htm`.

HERZBERG, A. AND GBARA, A. 2004. Protecting (even) naive Web users, or: preventing spoofing and establishing credentials of Web sites. Draft.

JIANG, S., SMITH, S., AND MINAMI, K. 2001. Securing Web servers against insider attack. In *the 17th ACSA/ACM Computer Security Applications Conference*, New Orleans, LA.

KAIN, K., SMITH, S., AND ASOKAN, R. 2002. Digital signatures and electronic documents: A cautionary tale. In *Advanced Communications and Multimedia Security*. Kluwer Academic, Norwell, MA.

LEFRANC, S. AND NACCACHE, D. 2003. Cut-&-paste attacks with JAVA. In *Information Security and Cryptology—ICISC 2002*. LNCS 2587, Springer-Verlag, Berlin.

MARCHESINI, J., SMITH., S., AND ZHAO, M. 2003. Keyjacking: Risks of the current client-side infrastructure. In *Proceedings of the 2nd Annual PKI Research Workshop*, Gaithersburg, MD.

MAREMONT, M. 1999. Extra! extra!: Internet hoax, get the details. *The Wall Street Journal*.

MOZILLA ORGANIZATION, THE. 2001. Gecko DOM reference. Online resource. `http://www.mozilla.org/docs/dom/domref/dom_window_ref.html`.

NORMAN, E. 2002. Personal communication.

PAOLI, F. D., DOSSANTOS, A., AND KEMMERER, R. 1997. Vulnerability of 'secure' web browsers. In *Proceedings of the National Information Systems Security Conference*.

PERRIG, A. AND SONG, D. 1999. Hash visualization: A new technique to improve real-world security. In *Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce*.

RESCORLA, E. 2001. *SSL and TLS: Designing and building secure systems*. Addison Wesley, Reading, MA.

ROME, J. 1995. Compartmented mode workstations. Online resource. `http://www.ornl.gov/~jar/doecmw.pdf`.

SECUNIA. 2004. Mozilla/mozilla firefox user interface spoofing vulnerability. Secunia Advisory SA12188. `http://secunia.com/advisories/12188/`.

SECURE SOFTWARE, INC. EGADS homepage. Online resource. `http://www.securesoftware.com/download_form_egads.htm`.

SMITH, S. 2000. *WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions*. Tech. Rep. IBM T.J. Watson Research Center Research Report RC 21851.

SMITH, S. 2001. WebALPS: A survey of e-commerce privacy and security applications. *ACM SIGecom Exchanges 2.3*.

SMITH, S. AND SAFFORD, D. 2001. Practical server privacy using secure coprocessors. *IBM Systems Journal 40*.

SULLIVAN, B. 2000. Scam artist copies payPal Web site. The page expired, but related discussion exists at `http://www.landfield.com/isn/mail-archive/2000/Jul/0100.html`.

TURNER, C. 2003. How do consumers form their judgments of the security of e-commerce web sites? In *ACM/CHI2003 Workshop on Human-Computer Interaction and Security Systems*, Fort Lauderdale, FL. `http://www.andrewpatrick.ca/CHI2003/HCISEC/index.html`.

TYGAR, J. AND WHITTEN, A. 1996. WWW electronic commerce and Java trojan horses. In *Proceeding of the 2nd USENIX Workshop on Electronic Commerce*.

UNITED STATES SECURITIES AND EXCHANGE COMMISSION. 1999. Litigation release no. 16266. Online Resource. `http://www.sec.gov/litigation/litreleases/lr16266.htm`.

WEISER, R. 2001. Personal communication.

WHITTEN, A. AND TYGAR, J. 1999. Why johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceeding of the 8th USENIX Security Symposium* (Washington D.C.).

YE, Z. 2002. Building trusted paths for Web browsers. M.S. Thesis, Department of Computer Science, Dartmouth College, Hanover, NH.

YE, Z. AND SMITH, S. 2002. Trusted paths for browsers. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, CA.

YE, Z., YUAN, Y., AND SMITH, S. 2002. *Web Spoofing Revisited: SSL and Beyond*. Tech. Rep. Department of Computer Science, Dartmouth College, TR2002-417.

YEE, K. 2002. User interaction design for secure systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, Singapore.

Received April 2003; revised May 2004; accepted August 2004