Trusted Paths for Browsers

Zishuang (Eileen) Ye Sean Smith
{eileen,sws}@cs.dartmouth.edu
Department of Computer Science
Dartmouth College

www.cs.dartmouth.edu/~pkilab/demos/spoofing/

June 12, 2002

Abstract

Computer security protocols usually terminate in a computer; however, the human-based services they support usually terminate in a human. The gap between the human and the computer creates potential for security problems. This paper examines this gap, as it is manifested in "secure" Web services. Felten et al demonstrated the potential, in 1996, for malicious servers to impersonate honest servers. Our recent follow-up work explicitly shows 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. This paper reports the results of our ongoing experimental work to systematically *defend* against Web spoofing, by creating a *trusted path* from the browser to the human user.

1 Introduction

In the real world, computer security protocols usually do not exist for their own sake, but rather in support of some broader human process, such as shopping, filing government forms, or accessing medical services. However, the computer science community, perhaps because of its training, tends to focus on the computers involved in these social systems. If, by exchanging bits and performing cryptographic operations, the client machine can correctly authenticate a trusted server machine and correctly reject an untrusted one, then we tend to conclude the system is secure.

This tendency overlooks the fact that, in such systems, the client machine may receive the information, but the human user typically makes the trust decision. Simply ensuring that the machine draws the correct conclusion does not suffice, if the adversary can craft material that nevertheless fools the human.

In this paper, we examine these issues as they relate to the Web. The security of the Web relies on *Secure Socket Layer (SSL)*—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. However, it is not the human user but the Web browser that carries out this protocol. After establishing the SSL channel, the Web browser displays corresponding signals on its user interface, such as locking the SSL padlock, changing the protocol header to *https*, and popping up warning windows to indicate that an SSL session has been set up. The human uses these signals to make his or her trust judgment about the server. The adversary can thus subvert the secure Web session simply by creating the illusion that the browser has displayed these signals.

The term *Web spoofing* denotes this kind of "smoke and mirrors" attack on the Web user interface. To defend against Web spoofing, we need to create 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.

1.1 Background: Effective PKI

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. 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) instantiation 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 [12, 21, 22, 23] examined the server end, and how

to extend the trust from the channel itself into data storage and computation at the server. 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 much broader implications (as Section 7.2 will discuss).

1.2 Prior Work

In their seminal work, Felten et al [10] 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 [5] also explored this area. (In Section 2.2, we discuss our work in this space.)

In related work on security issues of user interfaces, Tygar and Whitten examined both the spoofing potential of hostile Java applets [25] as well as the role of user interfaces in the security of email cryptography systems [27]. Work in making cryptographic protocols more tenable to the human—including visual hashes [18] and personal entropy [9]—also fits into this space.

The world of multi-level security [6] has also considered issues of human-readable labels on information. The *compartmented mode workstation (CMW)* [19] is an OS that attempts to realize this security goal (and others) within a modern windowing system. However, a Web browser running on top of CMW is not a solution for Web spoofing. 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 material from server and the material from the browser.

Although CMW itself is not a solution for Web spoofing, the approach CMW used for labeling is a good starting point for further exploration—which we consider in Section 4.1.

1.3 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. Section 3 develops criteria for systematic, effective solutions. Section 4 discusses some solution strategies we considered and the one we settled on, *synchronized random dynamic (SRD)* boundaries. Section 5 discusses how we implemented this solution and the status of our prototype. Section 6 discusses how we validated our approaches with user studies. Section 7 offers some conclusions, and discusses avenues for future work.

2 Web Spoofing

2.1 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. Can the human accurately determine the identity of the server with which their 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 [15, 20] is a good example)
- by *typejacking*—e.g., registering a hostname deceptively similar to a real hostname, offering malicious content there, and tricking users into connecting (the "PayPai" case [24] is a good example)

Furthermore, as is often pointed out [2], RFC 1738 permits the hostname portion of a URL to begin with a username and password. Hoke [20] could have made his spoof of a Bloomberg press release even more effective by prepending his IP-hostname with a "bloomberg.com" username. Most Web browsers (including the IE and Netscape families, but not Opera) would 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`.

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?

2.2 Our Initial Study

Felten et al [10] showed that, in 1996, a malicious site could forge many of the browser's UI 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 [5] 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 [29] whether and to what degree Web spoofing was still possible, with current technology.

Our experiment was more successful than we expected. To summarize our experiment, for Netscape 4 on Linux and Internet Explorer 5.5 on Windows 98, 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 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.

2.3 Overview of Techniques

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.

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 opening a new window with some of these elements turned off, and that did not work either. Finally, we tried opening a new window with *all* of the elements disabled—and that worked. 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 pre-caching to get these images to the user's machine before they're needed.

This pattern made us cautious about the effectiveness of simplistic defenses that eliminate some channel of graphical display.

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.

Our prior technical report [29] contains full technical details.

2.4 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 [7] 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 [1] 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 7.2, we revisit some of these issues.

3 Towards a Solution

Previous work, including our own, suggested some simplistic solutions. To address this fundamental trust problem in this broadly-deployed and service-critical PKI, we need to design a more effective solution—and to see that this solution is implemented in usable technology.

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 P, A, and the channel over which P was obtained. (For simplicity, we're ignoring things like the fact that multiple servers may be involved.)

We can think of the browser as executing two functions from this input space of Web page content and context:

- displaying sets of graphical elements in this window and others as *content* from the server
- displaying sets of graphical elements in this window and others as *status* about this server content.

Web spoofing attacks can work because no clear difference exists between the graphical elements of *status* and the graphical elements of *content*. There exist pages P_A, P_B from servers A, B (respectively) such that the overlap between *content*(P_A) and *status*(P_B, B) is substantial. Such overlap permits a malicious server to craft *content* whose display tricks users into believing the browser is reporting *status*.

To make things even harder, what matters is not the actual display of the graphical elements, but the display as processed by human perception. As long as the human perception of status and content have overlap, then spoofing is possible.

(Building a more formal and complete model of this problem is an area for future work.)

3.2 Trusted Path

From the above analysis, we can see the key to systematically stopping Web spoofing would be twofold:

- to clearly distinguish the ranges of the *content* and *status* functions, even when filtered by human perception, so that malicious collisions are not possible
- to 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 between users and the status component, that can not be impersonated by content component.

3.3 Design Criteria

We consider some criteria a solution should satisfy.

First, the solution should *work*:

- **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 7.2 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.
- **Effectiveness.** We need to ensure that the status information is provided in a way that the user can effectively recognize and utilize. For one example, the information delivered by images may be more effective for human users than information delivered by text. For another 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.

Secondly, the solution should be *low-impact*:

- **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.
- **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.

4 Solution Strategies

Having established the problem and criteria for considering solutions, we now proceed to examine potential strategies. Section 4.1 presents some approaches we considered but rejected; Section 4.2 presents the strategy we chose for our implementation. Table 1 summarizes how these strategies measure according to the above criteria.

4.1 Considered Approaches

No turn-off. As discussed above, one way to defend against Web spoofing is make it impossible for 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, this approach would overly constrict the display of server pages (many sites depend on pop-ups with server-controlled content) and still does not cover a broad enough range of browser-user channels. Furthermore, the attacker can still use images to spoof pop-up windows of his own choosing.

Customized content. Another set of approaches consists of trying to clearly label the status material. One strategy here would draw from Tygar and Whitten [25] 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, etc.) 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.

Meta-data titles. We considered having some meta-data, such as page URL, displayed on the window title. Since the browser sends the title information to the machine window system, the browser can enforce 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.

Meta-data windows. We considered having the browser create and always keep open an extra window just for meta-data. The browser could label this window to authenticate it, and then use it to display information such as URL, server certificate, etc.

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 meta-data. Furthermore, this approach would require a way to correlate the displayed meta-data 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 meta-data is referring.

As we will discuss shortly, CMW uses a meta-data window and a side-effect of Mozilla code structure forced us to introduce one into our design.

Boundaries. In an attempt to fix the window title scheme, we decided to 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.

CMW-Style Approach. CMW brought the boundary and meta-data window approaches together.

We noted earlier that CMW itself will not solve the spoofing problem. However, 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 meta-data window at the *bottom* of the screen, puts color-coded boundaries on the windows and a color (not text) in the meta-data window, and solves the correlation problem by having the color in the meta-data window change according to the security level of the window currently in focus.

The CMW approach inspired us to try merging the boundary and meta-data 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 meta-data 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 sepa-

ration 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's too late.

Because of these drawbacks, we decided against this approach. Our user study of a CMW-style simulation (Section 6) supported these concerns.

4.2 Prototyped Approach

We 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 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.

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 level.

This situation left us with a conundrum: the browser needs to mark trusted status content, but any deterministic approach to marking trusted content would be vulnerable to this image spoof. So, we need an automatic marking scheme that servers could not predict, but would still be easy and non-intrusive for users to verify.

Initial Vision. What we settled on was *synchronized random dynamic (SRD)* boundaries. 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.

The SRD solution would satisfy the design criteria:

- **Inclusiveness.** All windows would be unambiguously labeled as to whether they contained status or content data.
- **Effectiveness.** Like static colored boundaries, the SRD approach shows an easy-to-recognize security label at the same time as the content. Since a malicious server cannot predict the randomness, it cannot provide spoofed status that meets the synchronization.
- **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 others.
- **Minimizing intrusiveness.** By changing the window boundary but not internals, the server content, as displayed, is largely unaffected.

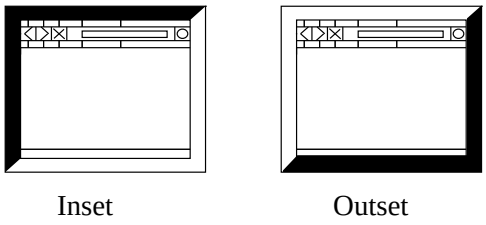

Figure 1 *Inset* and *outset* border styles.

In the SRD boundary approach, we do not try to focus so much on communicating status information as on distinguishing browser-provided status from server-provided content. The SRD boundary approach tries to build a trusted path 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.

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

We 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. As a consequence, all other windows stop responding and become inactive. This is because some modules are *singleton* services in Mozilla (that is, services that one global object provides to all threads in Mozilla). When one thread accesses such a service, all other threads are blocked. The Enter-SSL warning window uses the *nsPrompt* service which is one of the singleton services.

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 the 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 re-introduce a meta-data *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 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 non-textual 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.

Reality also introduced other semantic wrinkles, as discussed in Section 5.7.2.

5 Implementation

Implementation 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.

In Section 5.2 through Section 5.5 below, we discuss these steps. Section 5.7 discusses the current status of our prototype.

Figure 4 shows the overall structure.

5.1 Starting Point

In order to implement our trusted path solution, we need a browser as its base. We looked at open source browsers, and found two good candidates, Mozilla and Konqueror. Mozilla is the "twin" of Netscape 6, and Konqueror is part of KDE desktop 2.0. 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.

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 make it might be unnecessarily complicated for our purposes. Secondly, Mozilla is closely related to Netscape, which has a big market share on current desktops. Third, Konqueror only run on Linux; Mozilla is able to adapt to several platforms.

Additionally, although both of browsers are well documented, we felt that Mozilla's documentation was stronger.

5.2 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 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. Each XUL element is present as an object in Mozilla's *document object module (DOM)*.

Mozilla uses *Cascading Style Sheets (CSS)* to describe what each XUL element should look like. Collectively, this set of sheets is called a *skin*. Mozilla has customizable skins. Changing the CSS files changes the look-and-feel of the browser. (Figure 3 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.

(The new *chrome* feature introduces some wrinkles; see Section 5.7.2.)

5.3 Making the Boundaries Dynamic

We next need 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 different style.

To make window boundaries dynamic, we added a *borderStyle* attribute to the window.

```
externalwindow[borderStyle="true"]
{ border-style:  outset !important;}
```

When *borderStyle* is set, the boundary style is outset; when *borderStyle* is removed, the boundary style is inset. Mozilla observes the changes in attributes and updates the displayed *borderStyle* 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:

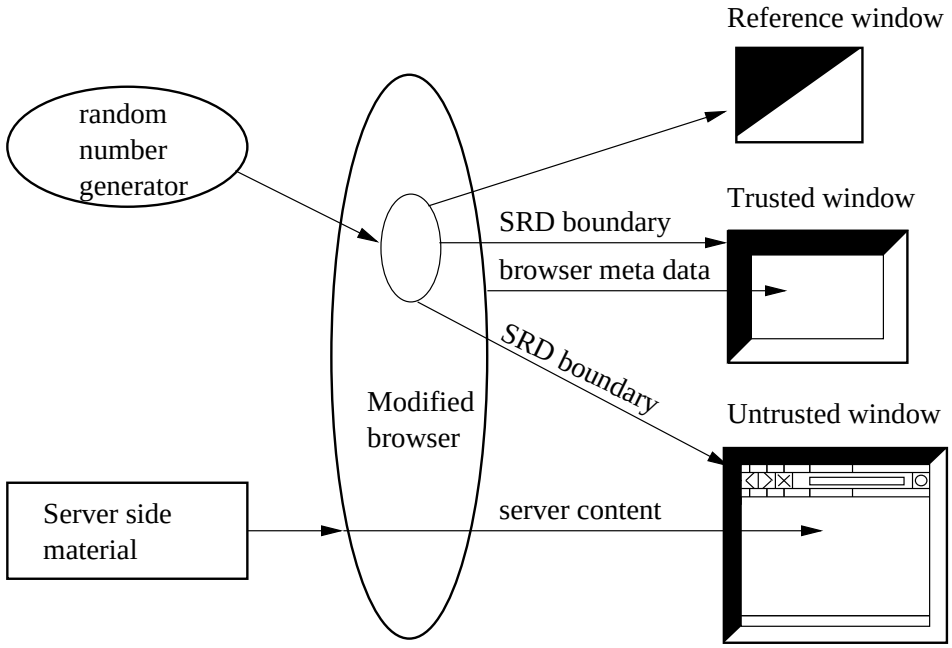

Figure 2 The architecture of our SRD approach.

	Inclusiveness	*Effectiveness*	*Minimizing User Work*	*Minimizing Intrusiveness*
No turn-off	*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***

Table 1 Comparison of strategies against design criteria.

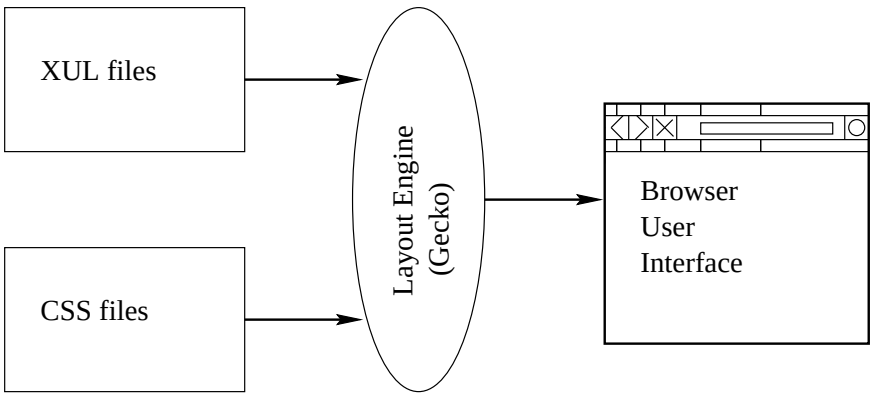

Figure 3 The layout engine uses XUL and CSS files to generate the browser user interface.

```
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("function name",
intervalTime)
```

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

```
<window id="example-window"
onload="setInterval(..)">
```

If the window element does not have an ID associated with it, we need to give it one in order to make the JavaScript code work. The JavaScript files need to include into corresponding *jar.mn* file in order to be packed into the same jar as the XUL file.

5.4 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 an *XPCOM* module to have them communicate to a single C++ object that directed the randomness.

XPCOM (the *Cross Platform Component Object Model*) 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 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 can not be set by JavaScript. The XPIDL compiler transforms this IDL 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.

5.5 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 adjustments 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.

5.6 Why This Works

This SRD approach works because:

- Server material has to be displayed in a window opened by the browser.
- When it opens a window, the browser gets to choose which type of window to use.
- Only the browser gets to see the random numbers controlling whether the border is currently inset or outset.
- Server content, such as malicious JavaScript, cannot otherwise perceive the inset/outset attribute of its parent window.

(Section 5.7.2 below discusses some known issues.)

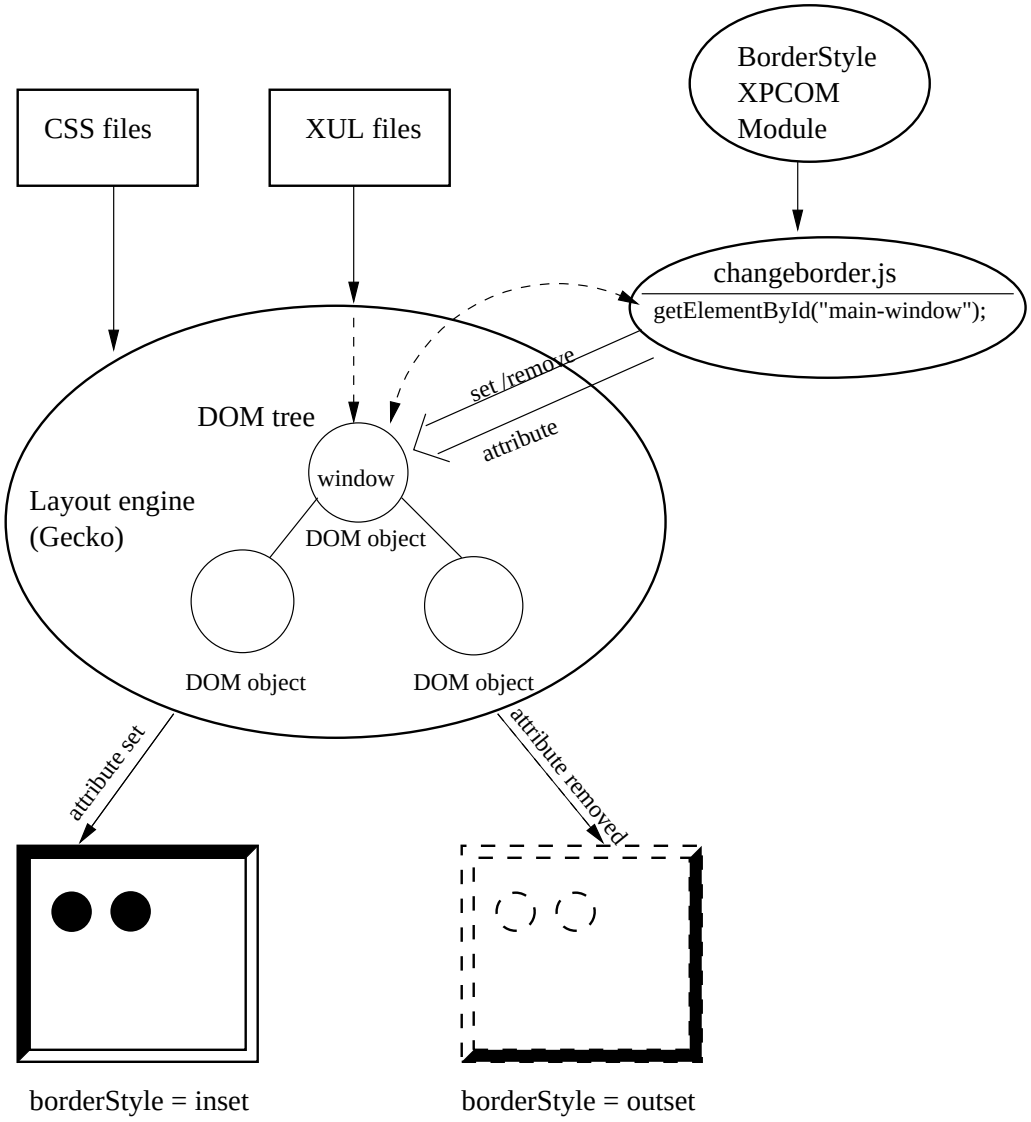

Figure 4 This diagram shows the overall structure of our implementation of SRD in Mozilla. The Mozilla layout engine takes XUL files as input, and construct a DOM tree. The root of the tree is the window object. For each window object, JavaScript reads the random number from borderStyle module, and sets or removes the window object attribute. The layout engine present the window object differently according to the attribute. The different appearances are defined in CSS files.

We elaborate on the last point above. The DOM is a tree-like structure to represent the document. Each XML element or HTML element is represented as a node in this tree. The DOM tree enables traversal of this hierarchy of elements. Each element node has DOM interfaces, which can be used by JavaScript to manipulate the element. For example, *element.style* lets JavaScript access the style property of the element object. JavaScript can change this property, and therefore change the element appearance.

When the Mozilla layout engine Gecko reads XUL files and renders browser user interface, it treats the window object as a regular XUL element, one DOM node in the DOM tree. Therefore, at the point, 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.

This root object has a child node, *document*. Under this *document* object, the server content DOM tree starts to grow. The root window does not provide the *window.style* interface. It also does not support any attribute functions [11]. Therefore, even though server-side JavaScript can get a reference of the window object, and call functions like *window.open*, it can not read or manipulate the window border style to compromise SRD boundaries. Our experimental tests also proved this statement.

5.7 Prototype Status

We have implemented SRD for the main navigator elements in modern skin Mozilla (currently Mozilla-0.9.2.1) for Linux. 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.

5.7.1 Inner SRD vs Outer SRD

In the current browsing paradigm, some otherwise untrusted windows, such as the main surfing window, contain trusted elements, such as Menu Bar, etc. 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 weakens 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.

5.7.2 Known Issues

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

Alert Windows. The only significant bug we currently know about pertains to alert windows. In the current Mozilla structure, *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. In our current implementation, the window boundary color is decided once, as "trusted". We are currently working with Netscape developers to determine how to have this code determine the nature of its caller and establish boundary color accordingly.

Signed JavaScript. Signed JavaScript from the server can ask for privileges to use XPConnect. 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.

Chrome feature. Mozilla added a new feature *chrome* to the *window.open* method. If a server uses the JavaScript

```
window.open("test.html",
"window-title", "chrome")
```

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, this new window cannot respond to the right mouse click and other reserved keystrokes, like *Alt+C* for copy under Linux. It is a known bug [4] that this new window cannot bring back the menu bar and the other bars, and it cannot print pages.

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 in ways that are bad for our purposes. So, we plan either to disable this feature, or to install SRD boundaries even on chromeless windows.

Pseudo-synchronization. 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-second interval. This imprecision is because only one thread can access the XPCOM module; all other threads are blocked until it returns. Since the 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.

6 Usability

The existence of a trusted path from browser to user does not guarantee that users will understand what this path tells them. In order to evaluate the usability of SRD boundary, we carried out user studies.

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.

6.1 Test Design

The design of the SRD boundary includes two parameters: the *boundary color* and the *synchronization*. They express different information.

- The boundary color indicates where the material comes from.
- The synchronization indicates whether the user can trust the information expressed by the boundary color scheme.

In our tests, we change the two parameters in order to determine whether the user can understand the information each parameter tries to express. We vary the boundary color over:

- trusted (orange)
- untrusted (blue)

We vary the synchronization parameter over:

- static (window boundary does not change)
- asynchronous (window boundary changes, but not in a synchronized way)
- synchronized

According to our semantics, a trustable status window should have two signals: a *trusted* boundary color, and *synchronized* changes. Eliminating the cases where the user receives *neither* of these signals, we have four sessions in each test: a static trusted boundary; a synchronized trusted boundary; a synchronized untrusted boundary; and an asynchronous trusted boundary.

We also simulated the CMW-style approach and examined its usability as well. In particular, the the CMW-style approach is less distracting than SRD boundary, because most of the labels are static. This reduces intrusiveness—but less distracting may also mean winning less attention.

We then ran three tests.

- In the first test, we turned off the reference window, and used only the SRD boundary in the main surfing window as a synchronization reference. We popped up the browser's certificate window with different boundaries, in four sessions.
- In the second test, we examined the full SRD approach, and left the reference window on, as a synchronization reference. We popped up the certificate window four different ways, just as in the first test. We wanted to see whether using reference window is helpful for providing extra security-related information, or whether it is needlessly redundant.
- In the third test, we simulated the CMW-style approach. Boundaries were static; however, a reference window always indicated the boundary color of the window to which the mouse points. 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.

  In the conventional CMW approach, the mouse has to be clicked on the window to get it focus at first. In our test, we used mouse-over, which gets the information to the user sooner. (In the future, we hope to

design more user studies to obtain additional data on how the time when status information arrives effect users' judgment during browsing.)

Before starting the tests, we gave the users a brief introduction about the SRD boundary approach. The users understood there were two parameters they needed to observe. The users also viewed the original Mozilla user interface, in order to become familiar with the buttons and window appearance. After viewing the original user interface, the users started our modified browser and entered an SSL session with a server. The users invoked the page information window, and checked the server certificate which the browser appeared to present. The page information window and the certificate window popped up with different boundaries, according to the session.

The users were asked to observe the windows for ten seconds before they answered the questions. The questions included what they observed of the two parameters of the window boundaries, whether they thought the window was authentic, and how confident they were about their judgment.

6.2 Users Description

We tried to collect users from different sophistication levels, in order to provide realistic results for evaluation of our design. More importantly, we wanted to collect information on how regular users recognize status from the browser user interface—this information would not only help us evaluate our current approaches, but could also potentially help drive designs of better user interfaces. (We see much potential future work here.)

For this user study, we had seven volunteers.

- Two can be ranked as experts: a scientist at Dartmouth's Institute for Security Technology Studies, and a Ph.D. candidate in computer science.
- Three are undergraduates who search the Web for information and buy products over the Web quite often. They also like new technology and are quick learners.
- Two are medical scientists who use Web mainly for searching research papers, and do not do online credit card transactions very often.

The user ages range from 21 to 40, covering the main age area of Web users. The users major in physiology, biology, computer science, engineering, psychology, sociology, medicine. Among these users, only the computer experts check the security features on their browsers before they submit credit card information online. Except for these two experts, only one subject had even heard of the phrase "SSL" and none of them knew what it meant. Except for the experts, no one checked the *https* and the lock icon.

6.3 User Study Results

We summarize the most significant results we observed from the tests.

6.3.1 No Reference Window

In the first test, we used dynamic boundaries but no reference window.

Response to the static trusted boundary. This is the first test session. When shown the certificate window with a static trusted boundary, only the computer experts correctly perceived the status information and asserted that the certificate window was not authentic. All the other users failed to make the correct judgment, although they were not confident about their decision. An interesting observation was that three out of five users who made wrong judgment at first, recalled the window in first session was inauthentic after finishing the first test. This shows how quickly the users can be educated.

Response to the synchronized trusted boundary. In this session, the users viewed the browser with proper SRD boundaries. Five out of seven users made the correct judgment. The ones who made the correct judgment were confident about the decision. The ones that failed to capture the right information were not confident about their decision.

Response to the synchronized untrusted boundary. In this session, the certificate window came with a blue (untrusted) boundary. Five out of seven users correctly recognized the certificate window was not authentic, because it should be in an orange boundary. The ones who made the correct judgment were confident about the decision, and thought the signal expressed by the color scheme was very clear.

Response to the asynchronous trusted boundary. In this session, the user needed to recognize that the trusted boundary was not changing correctly. All the users successfully judged this window was not authentic. They were also confident in their judgment.

This result surprised us: we thought the synchronization is not as strong a signal as the color. Apparently, human users recognize the synchronization parameters better than the color scheme. One reason may be that users pay more attention to dynamic features than to static ones. A second reason for this result may be that this is the last session of the first test. During the first three sessions, the

users may have learned how to observe and make judgment.

6.3.2 Full SRD

We then tested full SRD, with the reference window.

Response to the static trusted boundary. The reference window popped up before the main window started, which won most of the users' attention. Five out of seven users recognized the window status successfully. The ones made correct decision were confident about their decision.

Response to the synchronized trusted boundary. This time, all the users successfully recognized the status information and felt confident in their decision.

Response to the synchronized untrusted boundary. Six out of seven users made the correct judgment. They thought the signal expressed by the color was very clear.

Response to the asynchronous trusted boundary. All the users made the correct judgment. They all were confident about their decision, and thought the signals were very clear.

6.3.3 CMW-Style

In our last test, we simulated the CMW-style approach.

This test was an optional one for the users. Two out of four users who did this test successfully made the right judgment—but they were the experts. In general, the users felt confused about the information provided by the CMW reference window, and they tended to neglect it. We plan a more detailed study here.

6.4 User Study Conclusions

Different levels have very different responses. During our tests, we noticed that it was very obvious that the computer scientists have much faster reaction to security signals, and were more successful at recognizing what the signals meant. The other users took longer to observe the signals, and still did not always make the correct judgment. The user with the physiology background did not understand the parameters until the second session of the second test.

One conclusion is that computer scientists have a very different view of these issues from the general population. A good security feature may not work without good public education. For example, SSL has been present in Web browsers for years, and is the foundation of "secure" e-commerce, which many in the general public use. However, only one of our non-computer people heard of this phrase. Signals such as the lock icon—or anything more advanced we dream up—will make no sense to users who do not know what SSL means.

Users learn quickly. Another valuable feedback from our user study was that general users learned quickly, if they have some Web experience. Three out of five non-computer experts understood immediately after we explained SSL to them, and were able to perceive server authentication signals right away. The other two gradually picked up the idea during the one hour tests. At the end of the tests, all of our users understood what we intended them to understand.

This result supports the "minimal user work" property of our SRD approach: it easy to learn even for the people outside of computer science. The users do not do much work; what they need to do is observe. The status information reaches them automatically. No Web browser configuration or detailed techniques are involved.

Reference is better. Most of our users felt it was better to have the reference window, because it made the synchronization parameter easy to be observed. The reference window starts earlier than the main window, so it attracts user's attention. The users would notice the changing of boundary right after the main window starts up.

This result is ironic, when one considers that we only added the reference window because it was easier than rewriting Mozilla's thread code.

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 site use dynamic techniques. In our user study, most of the non-computer people did not even notice that a static window boundary existed in the first session test.

Automatic is better. The user study result from CMW-style approach simulation also indicates that indicating security information without requiring user action was better.

7 Conclusions and Future Work

7.1 Summary

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 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.

7.2 New Directions

This research also suggests many new avenues of research.

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 then construct "unnatural" Web pages with a variety of combinations of framesets, servers, 1x1-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 [3] 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. [17]

What is reported in the trusted path should also provide what the user needs to know to make a trust decision. For one example [8], 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 [12, 21, 22] 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 [26] and how they handle CA certificates with deliberately misleading names [17].

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.

Multi-Level Security. It is fashionable for younger scientists to reject the Orange Book and its associated body of work regarding multi-level 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 [18] in combination with various techniques, such as meta-data 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 [13], 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 help with our initial spoofing work; Denise Anthony and Robert Camilleri, for their 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, Mark Vilardo, Dan Wallach, Russell Weiser and the anonymous referees, for their helpful suggestions.

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.

A preliminary version of this paper appeared as Dartmouth College Technical Report TR2002-418.

References

[1] A. Alsaid, D. Martin. "Detecting Web Bugs with Bugnosis: Privacy Advocacy through Education." *2nd Workshop on Privacy Enhancing Technologies.* Springer-Verlag, to appear.

[2] R.J. Barbalace. "Making something look hacked when it isn't." *The Risks Digest*, 21.16, December 2000.

[3] S. Bonisteel. "Microsoft Browser Slips Up on SSL Certificates." *Newsbytes.* December 26, 2001.

[4] Bugzilla Bug 26353,"Can't turn chrome back on in chromeless window" `http://bugzilla.mozilla.org/show_bug.cgi?id=26353`

[5] F. De Paoli, A.L. DosSantos and R.A. Kemmerer. "Vulnerability of 'Secure' Web Browsers." *Proceedings of the National Information Systems Security Conference.* 1997.

[6] *Department of Defense Trusted Computer System Evaluation Computer System Evaluation Criteria.* DoD 5200.28-STD. December 1985.

[7] C. Ellison. "The Nature of a Usable PKI." *Computer Networks*. 31: 823-830. 1999.

[8] C. Ellison. Personal communication, September 2000. See `https://store.palm.com/`

[9] C. Ellison, C. Hall, R. Milbert, B. Schneier, "Protecting Secret Keys with Personal Entropy" *Future Generation Computer Systems.* Volume. 16, 2000, pp. 311-318.

[10] E. Felten, D. Balfanz, D. Dean, and D. Wallach. "Web Spoofing: An Internet Con Game." *20th National Information Systems Security Conference.* 1996.

[11] *Gecko DOM Reference.* `http://www.mozilla.org/docs/dom/domref/dom_window_ref.html`

[12] S. Jiang, S.W. Smith, K. Minami. "Securing Web Servers against Insider Attack." *ACSA/ACM Annual Computer Security Applications Conference.* December 2001.

[13] K. Kain, S.W. Smith, R. Asokan. "Digital Signatures and Electronic Documents: A Cautionary Tale." *Sixth IFIP Conference on Communications and Multimedia Security.* 2002. To appear.

[14] Konqueror. `http://www.konqueror.org/konq-browser.html`

[15] M. Maremont. "Extra! Extra!: Internet Hoax, Get the Details." *The Wall Street Journal.* April 8, 1999.

[16] The Mozilla Organization. `http://www.mozilla.org/download-mozilla.html`

[17] E. Norman (University of Wisconsin). Personal communication, April 2002.

[18] A. Perrig and D. Song. "Hash Visualization: A New Technique to Improve Real-World Security." *Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*. 131-138. July 1999.

[19] J. Rome. "Compartmented Mode Workstations." Oal Ridge National Laboratory. `http://www.ornl.gov/~jar/doecmw.pdf` April 23, 1995.

[20] *S.E.C. v. Gary D. Hoke, Jr.* Lit. Rel. No. 16266, 70 S.E.C. Docket 1187 (Aug. 30, 1999). `http://www.sec.gov/litigation/litreleases/lr16266.htm`

[21] S.W. Smith. *WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions.* Research Report RC 21851, IBM T.J. Watson Research Center, October 2000.

[22] S.W. Smith. "WebALPS: A Survey of E-Commerce Privacy and Security Applications." *ACM SIGecom Exchanges.* Volume 2.3, September 2001.

[23] S.W. Smith, D. Safford. "Practical Server Privacy Using Secure Coprocessors." *IBM Systems Journal.* 40: 683-695. 2001.

[24] B. Sullivan. "Scam artist copies PayPal Web site." *MSNBC.* July 21, 2000. (Now expired, but related discussion exists at `http://www.landfield.com/isn/mail-archive/2000/Jul/0100.html`)

[25] J.D. Tygar and A. Whitten. “WWW Electronic Commerce and Java Trojan Horses.” *The Second USENIX Workshop on Electronic Commerce Proceedings.* 1996.

[26] R. Weiser (DST). Personal communication, August 2001.

[27] A. Whitten and J.D. Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0.” *USENIX Security*, 1999.

[28] Z. Ye. *Building Trusted Paths for Web Browsers.* Master’s Thesis. Department of Computer Science, Dartmouth College. May 2002 (to appear).

[29] E. Ye, Y. Yuan, S.W. Smith. *Web Spoofing Revisited: SSL and Beyond.* Technical Report TR2002-417, Department of Computer Science, Dartmouth College. February 2002.