Dartmouth College
Dartmouth Digital Commons

Computer Science Technical Reports | Computer Science

7-1-2001

Web Spoofing 2001

Yougu Yuan
Dartmouth College

Eileen Zishuang Ye
Dartmouth College

Sean Smith Dartmouth College
false

Follow this and additional works at: https://digitalcommons.dartmouth.edu/cs_tr

Part of the Computer Sciences Commons

Dartmouth Digital Commons Citation

Yuan, Yougu; Ye, Eileen Zishuang; and Smith, Sean Dartmouth College, "Web Spoofing 2001" (2001). Computer Science Technical Report TR2001-409. https://digitalcommons.dartmouth.edu/cs_tr/185

This Technical Report is brought to you for free and open access by the Computer Science at Dartmouth Digital Commons. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Web Spoofing 2001

Yougu Yuan
yuanyg@cs.dartmouth.edu

Eileen Zishuang Ye
eileen@cs.dartmouth.edu

Sean Smith
sws@cs.dartmouth.edu

Department of Computer Science/Institute for Security Technology Studies *
Dartmouth College

Technical Report TR2001-409

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

July 2001

Abstract

The Web is currently the pre-eminent medium for electronic service delivery to remote users. As a consequence, authentication of servers is more important than ever. Even sophisticated users base their decision whether or not to trust a site on browser cues—such as location bar information, SSL icons, SSL warnings, certificate information, response time, etc.

In their seminal work on web spoofing, Felten et al [8] showed how a malicious server could forge some of these cues—but using approaches that are no longer reproducible. However, subsequent evolution of Web tools has not only patched security holes—it has also added new technology to make pages more interactive and vivid. In this paper, we explore the feasibility of web spoofing using this new technology—and we show how, in many cases, every one of the above cues can be forged.

In particular, we show how a malicious server can forge all the SSL information a client sees—thus providing a cautionary tale about the security of one of the most common applications of PKI.

We stress that these techniques have been implemented, and are available for public demonstration.

1 Introduction

Nearly every aspect of social, government, and commercial activity is moving into electronic settings. The World Wide Web is the de facto standard medium for these services. Inherent properties of the physical world make it sufficiently difficult to forge a convincing storefront or ATM that successful attacks create long-cited anecdotes (e.g., [7]). As a consequence, users of physical services—stores, banks, newspapers—have developed a reasonably effective intuition of when to trust that a particular service offering is exactly what it appears to be. However, moving from "bricks and mortar" to electronic introduces a fundamental new problem: bits are malleable. Does this intuition still suffice for the new electronic world? *When one clicks on a link that says "Click Here to go to TrustedStore.Com," how does one know that's where one has been taken?*

Answering these questions require examining how users make judgments about whether to trust a particular Web page for a particular service. Indeed, the issue of user trust judgment is largely overlooked; research addressing how to secure Web servers (e.g., [12, 13, 17]), how to secure the client-server connection (e.g., [10]), and how to secure

*This work was supported in part by by the U.S. Department of Justice, contract 2000-DT-CX-K001, and by Internet2/AT&T. However, the views and conclusions do not necessarily represent those of the sponsors.

client-side management (e.g., [15]) risk being rendered moot, if the final transmission of trust information the human user is neglected.

Users now routinely encounter Web (and Web-like) content as the alleged manifestation of some specific entity—a newspaper, a store, a government agency. How do users evaluate whether or not this manifestation is genuine? We briefly discuss some items that have re-surfaced recently.

- **Convincing HTML.** Once upon a time, e-mail consisted solely of text. However, mail tools have become sufficiently powerful that HTML attached to an email will be automatically rendered at the receiver's site. A malicious party can forge a news article by downloading legitimate HTML from a Web server, editing it, and attaching it to email.

  One might expect that recipients should know that Web pages should be viewed via Web transfers, not via email. However, law enforcement colleagues of ours report that this technique is sufficiently convincing to be a continuing source of problems.

- **Convincing URL.** More sophisticated users may base their decisions on the URL their browser tells them they are visiting. However, RFC 1738 permits more flexibility than many people realize. For a basic example, the hostname can be an IP address.

  As a case in point, in 1999, Gary D. Hoke [14, 21] created HTML that produced a realistic-looking Bloomberg press release pertaining to PairGain Technologies, Inc., posted this on `angelfire.com`, and posted a link to this page on a stock discussion bboard. Since the presence of `angelfire.com` instead of `bloomberg.com` in the URL might give away the hoax, Hoke disguised the link using by using Angelfire's IP address instead: [3]

  ```
  http://204.238.155.37/biz2/headlines/topfin.html
  ```

  Hoke's limited disguise was sufficient to drive the price of the stock up by a third (and also land him in legal trouble). However, the technology permits even more convincing disguises. For one thing, on many platforms (including all the Netscape installations we tested, except Macs), the IP address can be expressed as a decimal number (e.g., 3438189349) instead of the more familar slot-dot notation—further disguising things. Furthermore, (as pointed out yet again in [1]) RFC 1738 permits the hostname portion to include a username and password in addition to machine name. Hoke could have made his hoax even more convincing by finding (or setting up) a server that accepts arbitrary user names and passwords, and giving a URL of the form

  ```
  http://bloomberg.com:biz@3438189349/headlines/topfin.html
  ```

- **Typejacking.** URLs that appear more standard—but contain incorrect hostnames deceptively similar to hostnames users expect—is a trick long-used by many e-commerce vendors (primarily pornographers) to lure in unwitting customers.

  However, this technique can also be used maliciously. As we recently saw in the headlines [19], attackers established a clone of the `paypal` Web site at a hostname `paypai`, and used email to lure unsuspecting users to log in—and reveal their PayPal passwords. The fact that many mailers let a user simply click on a link, instead of pasting or typing it into a browser, and that in many fonts, a lower-case "i" is indistinguishable from a lower-case "l", facilitated this attack.

- **SSL to whom?** *Secure Sockets Layer (SSL)* is touted as the solution to secure Web connections and server impostors. However, even if a user can figure out how to examine the certificate testifying to a server's identity, it is not clear whether this information provides sufficient grounds for trust. For example [6], 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? (It turns out, yes. But how is the user to know?)

It is against this background that we started our work. The current visual signals that a browser provides do not appear to provide enough information for many trust judgments the user needs to make. But before we examine how to extend the browser user interface to handle things like security attribute certificates or delegation for servers, we need to examine whether the insufficient information the browsers do provide is reliable. The seminal web spoofing work of Felten et al [8] showed that, in 1996, a malicious site could forge many of these clues. Subsequent researchers [4]

reported difficulty reproducing these results. This scenario raises the question: *What's still spoofable today, in 2001?* Our answer: for standard browser installations, just about everything: including the existence of SSL sessions and the alleged certificates supporting them.

To return to our earlier question: how does one know the click has really taken one to TrustedStore.com? *One cannot.* On common Netscape and Internet Explorer platforms, if one surfs with JavaScript enabled, an adversary site can convincingly spoof the clues and appropriate behavior.

Section 2 establishes the background: the elements of Web pages, Web surfing, and advanced content tools. Section 3 discusses previous efforts in this area. Section 4 presents the sequence of techniques that we developed. Section 5 sketches some interesting extensions of these techniques. Section 6 concludes with some avenues for future work.

2 Background

Nearly all Internet users are familiar with the look and feel of Web pages and Web surfing. In this section, we quickly introduce the elements and the underlying technology, to establish a common terminology for the rest of the paper.

2.1 Anatomy of Browser Window

We start by quickly reviewing the elements of a browser window, using Netscape's "Classic" (e.g., 4.75) format. (See Figure 1.) The top line—with "File, Edit..." tags—is the *menu bar*. The next line—with "Back, Forward..." buttons—is the *tool bar*. Below the tool bar is the *location bar*, containing additional buttons as well as the *location line*, which displays the currently visited URL. Below the location bar is the *personal bar*. The main window is next; at the bottom, the *status bar* contains the *SSL icon*—which indicates whether or not an SSL session is established—as well as the *status line*, which indicates various status information, such as the URL associated with the hyperlink underneath the cursor. An *input field* is the portion of a Web page that collects input from users.

Interaction In a normal browser window, many of these elements are not static. Rather, user operations—such as moving a mouse over or clicking on an element—cause some corresponding reaction, such as displaying a *pop-up menu*. Typing into the location line and hitting return causes the browser to load that page.

Surfing In the typical act of browsing, the user selects (via clicking a link, or typing a URL) a Web site; that server returns an HTML file which the browser renders. Potentially, the file causes the browser to fetch and load other files from that server and other servers.

In a typical SSL session, the browser and server encrypt transmissions using shared session keys. The existence of an SSL session is indicated by the SSL icon; depending on configuration and platform, entering and exiting an SSL session (as well, potentially, other relevant SSL events) triggers the display of a *warning window*, usually with an "OK" or "OK, Cancel, Help" buttons.

2.2 Web Spoofing

For this paper, we offer this working definition of *web spoofing*: when malicious action causes the reality of the browsing session to differ significantly from the mental model a sophisticated user has of that session.

In our work, we confine this malicious action to content from a conspiracy of servers: e.g., the server that offers the fake "click here for TrustedStore.Com" link, and the server that offers the faked pages. In particular, we do not permit

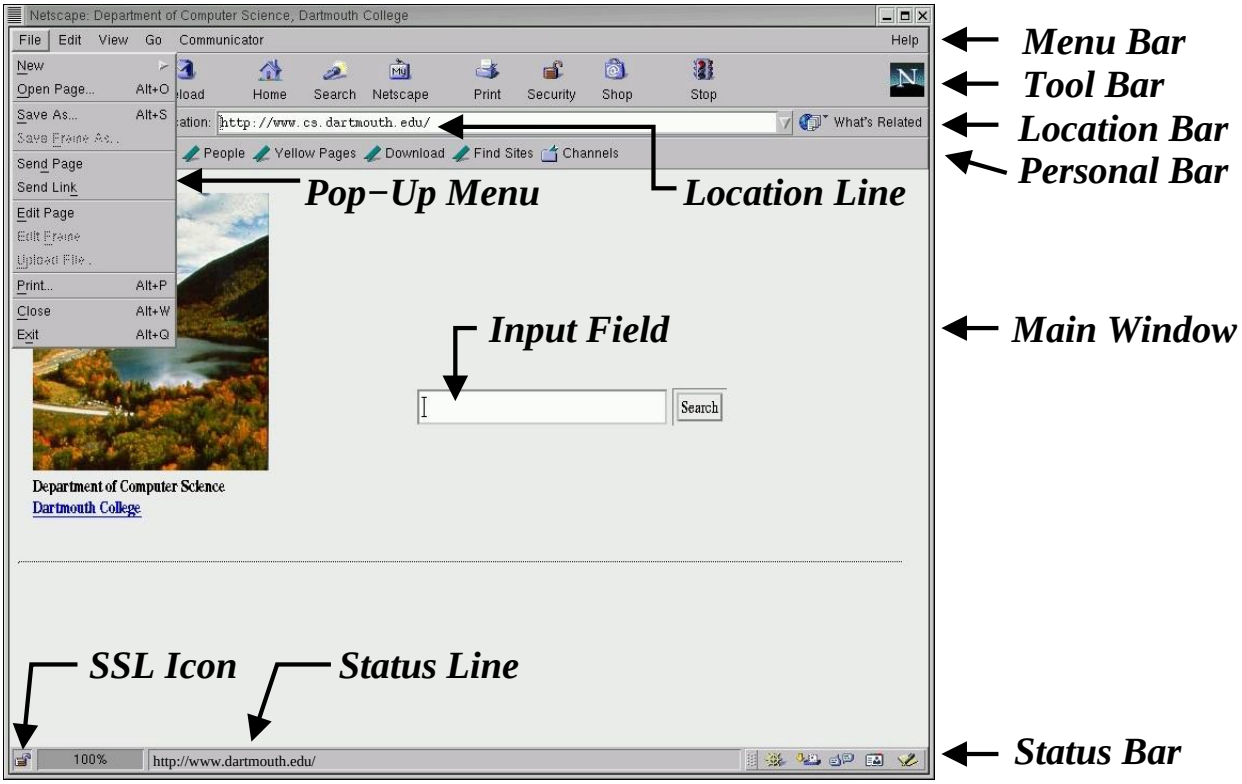

Figure 1 Elements of a typical browser window.

the adversarial servers to have signed scripts, and we will regard the the user's browser and computing platform to be out of the reach of an adversary. (Although, when one thinks of the many non-Netscape and non-Adobe sites that say "click here for the latest version of Netscape" or "click here to get Acrobat," Web spoofing could enable some very interesting platform attacks.)

In a typical Web spoofing attack, the attacker presents the client carefully designed web pages, and tricks the client to believe that he is communicating with a real server instead. Web spoofing is possible because the information which the server provides to the user is collected and displayed by the browser. Even in SSL sessions, the user judges whether a connection is secure based on information like the SSL icon, warning windows, location information, and certificates. But the user does not receive this information directly; rather, he draws conclusions from what the browser appears to tell him.

Advanced Content Tools The key to Web spoofing is the fact that simple "hypertext markup" (the "HT" of "HTML") has never been good enough. Web developers have continually wanted more interaction and flash than simple hypertext markup provides, and have developed a sequence of ways to embed executable content in Web pages. Currently, the primary parameters here are:

- the *language* of the executable,
- and the *location* where it runs.

For languages, *Java* and *JavaScript* are two that are commmonly used. For location, the primary choice is at the *client*, or at the *server*. Examples of server-side executables—acting under the requests coming from the client side (but not necessarily from the user) are *servlets* and *CGI* scripts. For client-side execution, *Dynamic HTML (DHTML)* [18, 5, 11] has come to be the widely-accepted catch-all term for using HTML, *Cascading Style Sheets (CSS)*, JavaScript, and other techniques to create vivid, interactive pages that depend primarily on client-side execution.

It is true that the emergence of executable Web content coincided with an increased focus on security—since, without further countermeasures, the act of browsing could cause an unwitting client to download and execute malicious code. However, this security work has been concerned with how to prevent malicious code from stealing sensitive information from the local disk or other web pages the client browses, and from tampering with the data that client has or shares with other servers. (For example, both Java applet and JavaScript by default have almost no access to local files.) Ironically, this security focus is completely inappropriate for Web spoofing: where the goal of the content is not to illegally modify client-side data, but rather to mislead the client, so that he/she gives the sensitive data out by mistake. Too often, traditional security models do not include user trust judgment!

3 Previous Work

Princeton As early as 1996, Felten et al at Princeton [8] originated the term *web spoofing* and explored spoofing attacks on Netscape Navigator and Microsoft Internet Explorer that allowed an attacker to create a "shadow copy" of the true web. When the victim accesses the shadow Web through the attacker's servers, the attacker can monitor all of the victim's activities and get or modify the information the victim enters, including passwords or credit card numbers. Source code is not available; according to the paper, the attack used JavaScript to rewrite the hyperlink information shown on the status bar; to hide the real location bar and replace it with a fake one that also accept keyboard input, allowing the victim to type in URLs normally (which then get rewritten to go the attacker's machine); and to replace the Document Source button the menu bar (to show the source the victim expects, not the real source). Apparently unable to spoof the SSL icon, the Princeton attack spoofed SSL by having the user open a real SSL session to the attacker's machine.

UCSB Subsequently, De Paoli et al at UCSB [4] tried to repeat the Princeton attack, but could no longer fake the location bar of Netscape Navigator using JavaScript.

However, De Paoli did show how the features in Java and JavaScript can be used to launch two kinds of attacks in standard browsers. One attack is using a Java applet to collect and send information back to its server. A client downloads a honey-pot HTML document that embeds a *spy applet* with its stop method overridden (although this enabling technique has since been deprecated). From then on, every time the client launches a *new* web page with an embedded Java applet, a new Java thread starts. But since the stop method of the spy applet is overridden, the spy thread continues running—and collects information on the new client-side Java threads, and sends them back to the attacker. The other attack uses a Java applet to detect when the victim visits a certain website, then displays an impostor page for that site in order to steal sensitive information, such as credit card number.

CMU In 1996, Tygar and Whitten from CMU [20] demonstrated how a Java applet or similar remote execution can be used as a *trojan horse*. The Java applet could be inserted into a client machine through a bogus remote page and pop up a dialog window similar to the true login windows. With the active textfield on the top of the image, the trojan horse applet would capture the keyboard input and transfer them to attacker's machine. Tygar and Whitten also gave a way to prevent these attack: window personalization.

None of these papers investigated how to *forge* SSL connections, when no SSL connection exists.

4 Our Techniques

Web browsers do not seem to provide sufficient information to enable trust judgments. However, attempting to remedy this shortfall begs the question: is the information they do provide trustworthy? Consequently, we postponed our long-term goal, and looked at how we might spoof using current Web technology.

4.1 The Scenario

Target The target we chose was *WebBlitz*, a web-based e-mail system used on our campus, and hosted by `basement.dartmouth.edu`. We chose this target simply because it has immediate familiarity to our local community (e.g., it is the primary e-mail tool for one of the authors), and because a successful spoof would demonstrate many interesting features: SSL, interaction, and sensitive data—WebBlitz userids and passwords are also good for many other academic transactions.

Three pages are relevant to our attack.

- a page with an *entry link* to the `basement` site;
- the (unsecured) welcome page at `basement`;
- the SSL-protected login/password page at `basement`

In normal operation, the entry link takes the user to the welcome page; the "WebBlitz" button on the Welcome page takes into an SSL session and the login page.

Language We chose JavaScript as our main tool for two reasons. First, JavaScript is fairly popular, with lots of sample code online that can used as resources. We want to show how easy this attack can be achieved without sophisticated techniques. Secondly, with the emergence of DHTML, supporting more powerful JavaScript functionality seems to be a trend of new browsers.

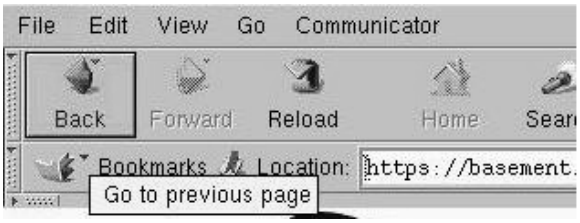

Figure 2 Sample fake tool bar pop-up, in Netscape.

4.2 First Attempt

In our first attempt, we tried to reproduce the old Princeton attack; like UCSB, we failed.

We found that we could not manipulate the location bar in the currently active browser window. (One suspects this may be the result of a security feature added in the wake of Princeton's work.) We can open a new window with the location bar turned off, but this creates new problems:

- 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).

4.3 Our Own Window

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`. 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 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.

4.4 Fake Interaction

To make the fake bars convincing, we need to 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.

Figure 2 and Figure 3 show samples of our fake tool bar interaction and a real tool bar interaction, for Netscape 4.75. (These pairs of figures ironically recall Borges [2].)

Figure 3 Sample true tool bar pop-up, in Netscape.

4.5 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, from Section 4.3, 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 contrained 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 multi-frame approach with one frame—containing a merged background image.

4.6 Location Bar Interaction

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's expected. The fake location bar can of course then 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 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 the 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 "use my fonts, not the document's," then the browser will ignore our specification and use the user's instead. We 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's not clear if it's available at all. This is an area for future work.

4.7 History Information

Another typical location bar behavior is displaying a pulldown menu of sites the user previously visited. (On Netscape, 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, 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.

4.8 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.

Unlike Princeton, we don't bother establishing a secure connection. 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 (that is, 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 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 succesful spoof. 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.

4.9 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. So, here's what we do:

- For 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.

  Initially, we noticed that that this fake warning window would not display properly, since it was covering genuine input fields.

  We originally addressed this problem by, in the spoofed warning window page, replacing these live input fields with an image that looks just like them.

  However, for some users, the redraw of the background image is noticeable; as an alternative, we tried simplying raising the fake warning window in a place that does not cover input fields. That works much better.

- For Internet Explorer, we use a Model 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. We are working on eliminating this.)

Figure 4 and Figure 5 show the fake and true warning windows for Netscape 4.75. (Again, we hope that Borges [2] would appreciate this.)

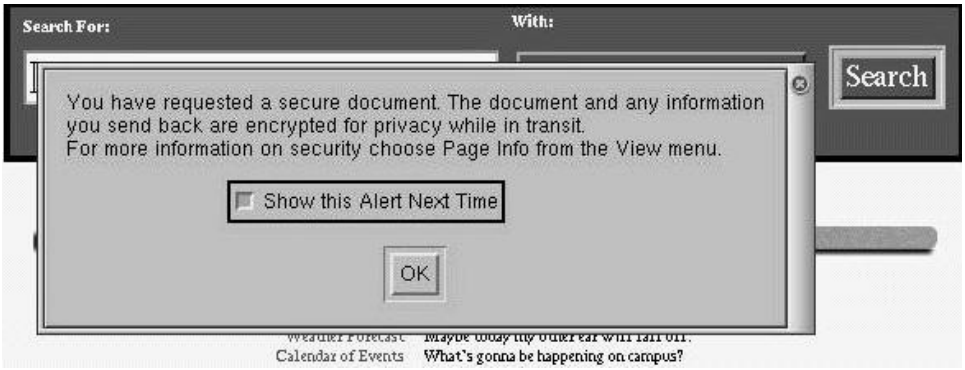

Figure 4 Our fake SSL warning window, in Netscape.

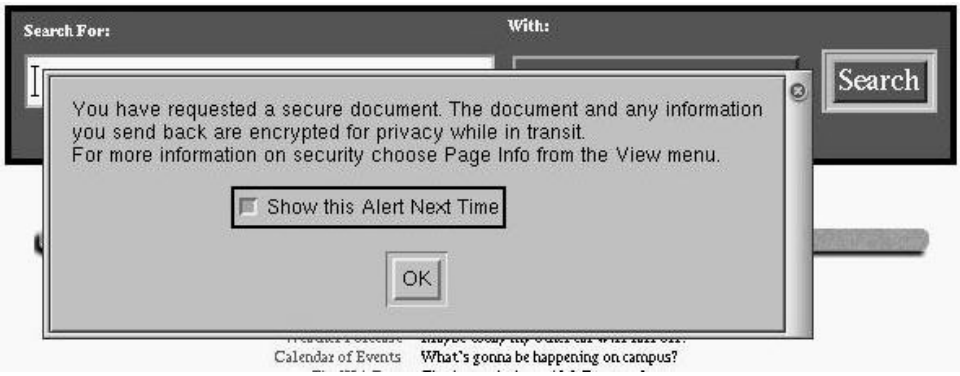

Figure 5 A true SSL warning window, in Netscape.

4.10 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, 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 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.

4.11 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 for user's browser, the onclick method will not be executed at all, so it just behave completely 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. (In theory, we should open a new window, to complete the spoof.) Otherwise the fake window will pop up.

4.12 Pre-Caching

In order to make the browser looks real, we use true browser images grabbed from screen (as noted earlier).

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 pre-caching. Each page pre-caches 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.

5 Extensions and Discussion

5.1 Information Collection

A critical part of making our spoof convincing is tuning the content to the user's browser. In the preceding discussion, for each element, we needed to consider how to do it both for Netscape and for Internet Explorer. Fortunately, the browser tells us this information, so we can tune the content automatically.

In theory, a spoofing server could collect even more detailed information. For example, we could query about the browser *version*. Version information can be useful because different version of browsers may have different look and functionality.

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 script," and we did not allow that for our attack. If we relax this restiction, 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. 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 "you are visiting our page with Netscape 6—would you like classic or modern format?" We would then provide the appropriate spoofed material.

Because our spoof techniques use a lot of images, it would be desirable for the attacker to know information about the bandwidth of the client's connection. We could achieve this goal by embedding some more code and a fair amount of images, video or audio data to the query page, using timing attacks [9] to check the loading time used by the client for that page, and using that data to estimate the bandwidth.

5.2 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 environment variables (e.g., window size) could be queried dynamically to make the fake window fit more easily in the environment.

5.3 Minimum Exposure

A cautious adversary might worry that the more one cheats, the more likely one will be found out. Given our current techniques, this would be good for a spoofer to keep in mind.

In our attack, we limit the clients to those using common browser/OS configurations that we can handle; we send the rest to the real page. Some other cautionary tricks that could be applied include:

- In the CGI or any other program that receives and logs the stolen data, we could put a trigger that replaces all the fake pages with good ones that direct the user to the real site.
- It's not trivial to make the fake window as fully functional as the real browser window. So, when some implementation gets really complicated, we could put a trap there that closes the current window—making it look like a bug that crashes the browser. (This is still "normal" behavior of modern browsers.)

5.4 Netscape vs. Microsoft

As we mentioned, we developed spoofing attacks for Netscape Navigator 4.75/4.76 on Linux and Internet Explorer 5.5 on Windows 98; we have been exploring Netscape 6. When we set up the experimental attack for these two families, we have fairly different experiences.

Table 1 summarizes the relevant differences we encountered (but note that our intention is not to suggest that one browser is more "better" than the other). On one hand, a browser can make it easier to implement some nice features for Web design. On the other hand, this convenience also makes spoofing easier.

As new updates emerge, the trend of the browser development is to give better support to script language. This trend makes the spoofer's job easier. For example, useful spoofing features like *popup objects* are supported by Internet Explorer 5.5 but not even Internet Explorer 5; as noted earlier, `innerText` is not fully supported until Netscape 6.

- **Popup Objects** Only supported by Internet Explorer and only since 5.5, the popup object feature greatly facilitates the making of menus and dialog-like content. Netscape 4 has a less flexible feature called *layer*. Netscape 6 abandoned some features of layer, but the same functionality can be provided by other new APIs.
- **Dialog** Internet Explorer has direct support to Modal/Modalless dialogs, but it appends a notice on its title bar, which can be observed by careful user. For Netscape, there is no direct support.
- **Dynamically Resizable Tables.** One of the keys to making a *resizable* fake window is to make some part of the display dynamically change its size according to the current window size. This feature is directly supported in Internet Explorer since version 5; we have not yet figured out to implement it for Netscape.
- **Dynamically Editable HTML Content.** This feature could also be used for an editable fake location line. As noted earlier, the `<input>` tag is problematic because we do not know a good way to change the size of the input dynamically according to the window size.

6 Conclusions and Future Work

In 2001, the original Princeton attacks may not be directly reproducible; but inspired by them, we have shown even more extensive spoofing is now possible.

Research in computer science, perhaps more than other fields, requires saying things in the language of each component of one's audience. One component is only convinced by seeing real code work. We have done that.

	Netscape 4.75	*Internet Explorer 5.5*
Query OS/browser?	Yes	Yes
Query/Change preferences?	Yes, with signed script	No
Dialog Boxes?	Yes, with large amount of DHTML coding	Yes, directly supported
Pop-up Windows ?	Yes, but inflexible (the *layer* feature)	Yes, flexible and easy (the *popup object*)
Resizable Tables?	Maybe. (With event triggers and window size queries?)	Yes, directly supported
Dynamic Editable HTML content?	No.	Yes, well supported.

Table 1 Comparison of some spoof-enabling features of popular browsers.

6.1 Putting it All Together

To summarize our experiment: for Netsape 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.
- If the user clicks on this link, and either does not have JavaScript enabled, or is using a browser/OS combination we do not support, then they really will go to that site.
- Otherwise, their browser opens a new window that appears to be a fully functional browser window, at that site. Clues, bars, location information, and most browser functionality all appear correct for that site. Except, the user is not visiting that site at all; he is visiting ours.
- Furthermore, if the user clicks on a "secure" link from this site, we can make convincing SSL warning windows appear, then lock the SSL icon, and have the SSL certificate information all appear as the user expects—except no SSL connection exists, and all the user's "secure" information is being sent in plaintext to us.

6.2 Future Spoofing Work

Our fake Web pages are not perfect. In our demonstration, we only implemented enough to prove the concept; however, as noted earlier, we are not yet able to forge some aspects of legitimate browser behavior:

- Creating convincing editable location lines appears to depend on the user's font preferences, which we cannot yet learn. Either we gamble, or we do not have editable lines.
- We cannot yet obtain the user's genuine history information for the pulldown history options.
- If the user resizes our fake Netscape windows, the content will not behave as expected.
- As Netscape 6, with its modifiable formats, grows in popularity, we need to examine how to provide spoofed material that either matches the user's format, or does not cause undue alarm.

6.3 Countermeasures

As Tygar and Whitten point out, customization is an effective way to prevent web spoofing. Although unsigned JavaScript can detect the platform and browser the client is using, it cannot (yet) detect the detailed window setting which may affect the browser display.

Felten et al suggested disallowing Web content to remove the location bar, ever. This technique would also be effective, but only when the client pays attention to the content inside the location bar.

As part of on-going research [22], we are currently examining other avenues for providing an unspoofable channel for the browser to communicate to the user, and for structuring communications here in a way users can understand.

Our initial motivation was not to attack but to defend: to "build a better browser" that, for example, could clearly indicate security attributes of a server (and so enable clients to securely use our server-hardening techniques [12, 13, 17]).

We hope to use our understanding of spoofing to build something that might be unspoofable.

References

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

[2] Jorge Luis Borges. "Pierre Menard, Author of the *Quixote*." *Labyrinths: Selected Stories and Other Writings.* New Directions, 1964.

[3] M. Broersma and L. Barrett. "Bogus Report Boosts Internet Stock." *ZDNet UK.* April 8, 1999.

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

[5] *Dynamic HTML Lab.* `http://www.webreference.com/dhtml/`.

[6] Carl Ellison. Personal communication, September 2000.

[7] "Fake ATM Machine Steals PINS." *The Risks Digest.* 14:59, May 1993.

[8] E. W. Felten, D. Balfanz, D. Dean and D. Wallach. *Web Spoofing: An Internet Con Game.* Technical Report 54-96 (revised). Deptarmtent of Computer Science, Princeton University. February 1997.

[9] Felton and Schneider. "Timing Attacks on Web Privacy." *ACM CCS*. 2000.

[10] Alan O. Freier, Philip Karlton, Paul C. Kocher. "The SSL Protocol Version 3.0." November 18, 1996. `http://home.netscape.com/eng/ssl3/draft302.txt`

[11] Danny Goodman. *JavaScript Bible.* 4th Edition. Hungry Minds, Inc., 2001.

[12] S. Jiang. *WebALPS Implementation and Performance Study.* (Master's Thesis). Computer Science Technical Report TR2001-399, Dartmouth College. June 2001.

[13] S. Jiang, S.W. Smith, K. Minami. *Securing Web Servers against Insider Attack.* Computer Science Technical Report, Dartmouth College. To appear, July 2001.

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

[15] The Mozilla Organization. *Personal Security Manager (PSM).* `http://www.mozilla.org/projects/security/pki/p`

[16] Netscape. *JavaScript Sample Code.* `http://developer.netscape.com/docs/examples/javascript.html`

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

[18] D. Steinman. *Dynamic Duo: Cross-browser Dynamic HTML.* `http://www.dansteinman.com/dynduo/`

[19] Bob Sullivan. "Scam artist copies PayPal Web site." *MSNBC.* `http://www.msnbc.com/news/435937.asp?cp1=1#BODY` July 21, 2000.

[20] J. D. Tygar and Alma Whitten. "WWW Electronic Commerce and Java Trojan Horses." *The Second USENIX Workshop on Electronic Commerce Proceedings.* 1996.

[21] United States Securities and Exchange Commission. *Securities and Exchange Commission v. Gary D. Hoke, Jr.* Litigation Release No. 16266. August 30, 1999.

[22] E.Z. Ye. *Securing Browsers.* Master's Thesis, Department of Computer Science, Dartmouth College. In progress.