2-4-2002

Trusted Paths for Browsers: An Open-Source Solution to Web Spoofing

Zishuang (Eileen) Ye
Dartmouth College

Sean Smith
Dartmouth College

Trusted Paths for Browsers: An Open-Source Solution to Web Spoofing

Zishuang (Eileen) Ye, Sean Smith
Department of Computer Science
Dartmouth College

Technical Report TR2002-418

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

February 4, 2002

Abstract

The security of the vast majority of "secure" Web services rests on SSL server PKI. However, this PKI doesn't work if the the adversary can trick the browser into appearing to tell the user the wrong thing about the certificates and cryptography. The seminal web spoofing work of Felten et al [6] demonstrated the potential, in 1996, for malicious servers to impersonate honest servers. Our recent follow-up work [15] 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 work to systematically *defend* against Web spoofing, by creating a trusted path from the browser to the user. Starting with the Mozilla source, we have implemented techniques that protect a wide variety browser-user communications, that require little participation by the user and minimal disruption of the displayed server content. We have prepared shell scripts that install these modifications on the Mozilla source, to enable others to replicate this work.

In on-going work, we are cleaning up and fine-tuning our code. In future work, we hope to examine more deeply the role of user interfaces in enabling users to make effective trust judgments.

1 Introduction

Section 2 discusses the problem. PKI (or any other distributed security scheme) needs to address not just the channel from server to client, but also from client to human. Web spoofing techniques can subvert this last step.

Section 3 develops criteria for a systematic, effective solution, that secures as broad as possible family of parameters that humans use to form trust judgments about servers. Section 4 discusses some solution strategies we considered and the one we settled on, *synchronized random dynamic boundaries*. Section 5 discusses how we implemented this solution and the status of our prototype. Section 6 offers some conclusions, and discusses avenues for future work.

2 The Problem

2.1 Effective Trust Judgments

The research 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 decided instead to focus on a broader definition of "infrastructure" as "that which is necessary for public-key cryptography to achieve this vision in practice." We then focused on the Web server 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 (e.g., [7, 11]) examined the server end, and how to extend the trust from the channel itself into data storage and computation at the server.

However, computer scientists tend to think of computation and communication as ending with a machine. But in the case of the Web, the true "client" is not the browser, but is rather the human using the browser. For the "secure Web" to work, the *human* needs to be able to make effect trust judgments about the server with which his or her browser is interacting. Our immediate motivation was that, for our server-hardening techniques to be effective, the human needs to determine if the server is using them; however, this issue has much broader implications.

2.2 Web Spoofing

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 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 (e.g. [9, 14])
- by *typejacking*—e.g., registering a hostname deceptively similar to a real hostname, offering malicious content there, and tricking users into connecting (e.g., [12])

Furthermore, (as pointed out yet again in [1]) RFC 1738 permits the hostname portion of a URL to begin with a username and password. Many servers (including ours) ignore these; Hoke [14] could have made his spoof of a Bloomberg press release even more effective by prepending his IP-hostname with a "bloomberg.com" username. (See our web site for a demonstration.)

More sophisticated 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 examine the server's certificate and validation information, to make full use of the server PKI.

The seminal web spoofing work of Felten et al [6] showed that, in 1996, a malicious site could forge many of the clues 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.) Subsequent researchers [3] reported difficulty reproducing these results, and Web techniques and browser user interface implementation has evolved a lot since 1996.

In our initial study [15], we examined whether (and to what degree) Web spoofing was still possible, with current technology. 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 mouseover, 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 functional browser window, at that site. Clues, bars, location information, and most browser functionality can be made to appear correct for that site. However, 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.

A demonstration is available at our web site, and our prior technical report [15] contains full technical details. (In a nutshell, we carefully examined how, for each platform, to provide server content that, when rendered, would appear to be the expected element. Since the browser kindly tells us its OS and software, we can customize the response appropriately.)

2.3 Other Factors

However, our goal was enabling users to make effective trust judgments about a server. The above spoofing techniques focused on server *identity*. As some researchers (e.g., [4]) observe, identity is just one component for such a judgment—usually not a sufficient component, and arguably not a necessary component. We revisit this question in Section 6.2.

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'll start with a slightly simplified model.

Browsers render content. When a browser is directed to a site (by a user, or by previous content), content C is downloaded and rendered, and the user perceives some set

$$\textit{stuff}(C) = \textit{status}(C) \cup \textit{content}(C)$$

of windows, boxes, and elements; data from the server, and meta-data from the browser.

The intention is that users can carry out a trust evaluation in two steps:

- They can recognize the material that is status information:

$$\textit{recog}(\textit{stuff}(C)) = \textit{status}(C)$$

- They can then judge the content based on an evaluation of that status information:

$$\textit{judge}(\textit{stuff}(C)) = \textit{eval}(\textit{recog}(\textit{stuff}(C)))$$

Web spoofing attacks work because the *status* and *content* functions that browsers typically implement suffer from collisions: there exist C_1, C_2 such that $\textit{content}(C_2) \cap \textit{status}(C_1)$ can be substantial. This overlap permits a malicious server to provide content C_2 which tricks users into recognizing as status from C_1:

$$\textit{recog}(\textit{stuff}(C_2)) = \textit{status}(C_1)$$

Consequently, users mistakenly judge C_2 as they would C_1.

3.2 Trusted Path

A key to systematically stopping Web spoofing would be to modify browsers so that their *status* and *content* functions do not have such collisions, and so users can (in theory, at least) always perform *recog*, and hence *judge*, correctly.

In some sense, this is the classic *trusted path* problem. The browser software becomes a TCB, we need to establish a trusted path between users and the *status* component, that cannot be impersonated by *content* output.

(Whether users should in fact trust their browsers is another story, especially given the proliferation of "click here for the latest Netscape" links one finds on non-Netscape sites.)

3.3 Design Criteria

What are the criteria a solution must satisfy?

- **Effectiveness.** We need to ensure that users can correctly recognize as large a subset of *status* as possible. Browsing is a rich experience; lots of parameters play into user trust judgment (and, as Section 6.2 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.

- **Work.** We cannot expect users to do a lot of work. This constraint eliminates the clearly impractical cryptographic approach of having the browser digitally sign each trusted document element. This constraint also eliminates more practical schemes where users set up customized, unguessable skin preferences.

- **Intrusiveness.** We must minimize our intrusion on *content*: on how documents from servers (and browsers) are displayed. The web is an important vehicle for commerce and services; an industry exists in optimizing web design to achieve various goals.

  This constraint eliminates the simplistic solution of turning off Java and JavaScript.

  This constraint also eliminates the Tygar-Whitten approach [13] of customizing backgrounds.

In a nutshell, our solution must be *effective* (for "trust judgment" functions that are still evolving), and be *low-impact*.

Implementability. In the long run, an effective solution must be real. Examining the options deploying our solution, we decided that working within the open-source code base of Mozilla [10] was the most feasible (although we had also given serious consideration to Konqueror [8]). This gave rise to another constraint: it must be possible for our solution to be prototyped—and replicated—as modifications to the Mozilla base.

4 Solution Strategies

4.1 Rejected Approaches

One approach to defending against some of the techniques we used in our spoofing work would be to identify page elements with high-trust information or functionality (such as the Menu Bar, Tool Bar, Status Bar, etc.), prevent them from being turned off in the current browser window, and prohibit the opening of new windows with them turned off. However, we felt this technique would not cover a broad enough range of browser-user channels (what about pop-up warning windows, or certificate information pages?), and would also overly constrict the the display of server pages.

A natural way to address these broader problems would be to clearly label the *status* material, in way that distinguishes it from *content* material. We considered and rejected several such approaches.

In one such rejected approach, we considered having the user enter a "MAC phrase" of their own choosing at start-up. The browser could then insert this MAC phrase into each *status* window (certificate status, SSL warning boxes, etc), to authenticate it. However, we decided that this required too much work from the user.

In another rejected approach, we considered adding such meta-data ("trusted status" or "something else") to the title information that Mozilla sends to the machine's windowing system. However, we did not really believe that users would pay attention to these title bars; furthermore, a malicious server could still spoof such a window by offering an image of one within the regular content.

In an attempt to fix the above 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. (This is how we spoofed SSL warning windows in our earlier work.)

4.2 Synchronized Random Dynamic Boundaries

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

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. At random intervals, the browser would change the styles on all its windows. The SRD solution would satisfy the design criteria:

- **Effectiveness**: A malicious server that did not know the randomness could not provide content that changed at the right time; by the flashing, users could recognize *all* windows that came from the browser; and by the color, users could distinguish those containing pure *status* information.
- **Work**: To authenticate a window, all a user would need to do is observe whether its border is changing in synch.
- **Intrusiveness.** By changing the window boundary but not internals, server content, as displayed, is largely unaffected.

Some browser windows, like the main window, contain trusted elements (such as the Menu Bar, etc) as well as an area for rendering untrusted material from the server. As far as we could tell in our spoofing work, untrusted material could not overlay or replace these trusted elements (rather, our attack worked by opening new windows without these elements, and adding our own facsimiles). The SRD approach thus leads to a design question:

- Should we just mark the outside boundaries?
- 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* 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 an trusted SRD boundary to be trusted depends on the boundary of its window) needlessly and perhaps dangerously complicate the user's participation?

The reality of implementation required modification to these semantics, as Section 5.4 discusses.

5 Implementation

We prototyped the SRD-boundary solution using Mozilla open source on Linux.

Implementation took three 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.

In Section 5.1 through Section 5.3 below, we discuss these steps. Section 5.4 discusses implications the implementation has for the semantics of our solution. Section 5.5 discusses the current status of our prototype.

5.1 Adding Colored Boundaries

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; this set of sheets together is called a *skin*. Mozilla has customizable skins, which are CSS files located in the source code "themes" directory. Changing these CSS files changes the look-and-feel of the browser.

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 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. In browsing, only the "navigator window" contains material from the server; the others (page info, SSL warning, etc) contain trusted material from the browser.

5.2 Making the Boundaries Dynamic

Window objects can have *attributes*. When the attibute is set, the window can be displayed with different style. This different style also is defined in global.css file.

To make window boundaries dynamice, we added a "borderStyle" attribute to the window.

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

When borderStyle is true, the boundary style is *outset* (meaning: the left and top are dark, and the bottom and right are light); when borderStyle is "removed" (i.e., false), the boundary style is *inset* (vice-versa). Mozilla itself has an application shell that implements an *observer* that notices changes in attributes and updates the displayed borderstyle accordingly.

With a reference to a window object, JavaScript code can automatically set the attribute and remove the attribute associated with that window. We use the

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

method to get the reference.

When the window's attribute is changed by JavaScript code, the browser observer object notices the change and schedules a browser event. The event is executed and repaints the boundary with different style.

Each XUL file links to JS 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 JS file and linked it into each corresponding XUL file.

For example, the file `example.xul` may have a window element defined as

```
<window id=" example-
window">...  </window>
```

Then the JavaScript file named `example-changeBorder.js` may have code

```
document.getElementById("example-
window")
```

The link inserted into the `example.xul` is

```
<script type="application/x-
javascript"
src="chrome://$CHROMPATH/example-
changeBorder.js''>
```

(The `CHROMPATH` depends on the directory in which the JS file resides.)

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 associate with it, we need to give it one in order to make the JavaScript code work. The JS files need to include into corresponding `jar.mn` file in order to be packed into the same jar as the XUL file.

5.3 Adding Synchronization

All the JavaScript files need to look at the same random number, in order to make all windows flashing 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 is the basic element of Mozilla structure. Its interface is written in *Cross-Platform Interface Description Language (XPIDL)*.

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 en-

ables 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, which only has a string that is read-only (from the JavaScript point of view). The XPIDL compiler transforms this IDL into a header file nsIBorderStyle.h and a typelib file nsBorderStyle.xpt. The nsIBorderStyle has an interface for a public function, GetValue, which can be called by Mozilla JavaScript through XPConnect. The NsBorderStyleImp class implements the nsIBorderStyle 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.

(We use these polling techniques since more advanced synchronization primitives did not easily integrate with Mozilla's JavaScript.)

5.4 Semantic Wrinkles

This implementation work led to changes to the SRD semantics as we originally envisioned.

One of the first changes is the precision of synchronization. It turned out that coordinating all the SRD boundaries to change at precisely the same real-time instant was not feasible within the current codebase. Instead, in our current implementation, the changes all happen within an approximately 1-second window. 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.

A more substantial issue is the fact that, due to the Mozilla structure, the existence of an SSL Warning Window blocks the threads associated with maintaining other window activity—including their SRD boundaries.

The default approach (taken in our current prototype) is to declare that a trusted SRD window, when all boundaries freeze, is trusted. However, this approach is troubling for two reasons.

First, such windows still might be forgable. Perhaps a server could 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.

More seriously, such windows greatly complicate the semantics of how the user decides whether to trust a window.

Currently, we are exploring how, within the Mozilla source, to keep at least one other window—perhaps a special reference window (trusted) opened at browser start-up—active at all times, even with SSL warning windows. Minimally, we can do this by having the XPCOM module fork a separate process that controls its own X-window.

5.5 Prototype Status

As with our earlier spoofing work, we feel that a simple paper discussion of ideas is not sufficient. If we are going to make an impact, we need to make these ideas work in the field.

As discussed in Section 5, we have implemented outer-SRD for the main navigator elements in modern skin Mozilla 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, as well as an animated gif giving the look-and-feel of browsers enhanced with outer-SRD and inner-SRD, are available now by request, and should be up on our web site very shortly.

In the lab, we are finishing modifications to add outer SRB to the remaining elements (PSM, mail, etc.,) as well as preparing an alternative version that adds inner SRB. We are also finishing implementation of the reference-window approach. We anticipate that this work should be done within two months.

6 Conclusions and Future Work

6.1 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 minimize work by the user and intrusiveness in how server material is rendered, and be deployable within popular browser platforms.

We believe our SRD solution meets this criteria. The major work remaining is to decide how to handle the background-freeze issue discussed above, and to decide whether inner-SRD enhances or weakens usability, and if this enhancement outweighs the increased intrusiveness.

We also offer this work back to the community, in hopes that it may drive more thinking (and also withstand further attempts at spoofing).

6.2 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 [2] that many IE platforms actually use different validation rules on some instances of these multiple SSL channels.

What is reported in the trusted path should also provide what the user needs to know to make a trust decision. For one example [5], 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 [7, 11] work) suggests that a user might want to know properties

in addition to server identity. Perhaps the trusted path should also handle attribute certificates.

User Studies. The existence of a trusted path from browser to user does not guarantee that users will understand what this path tells them. It would be interesting to do a follow-on user study, exploring how humans make judgments about servers how usable our approach is, and how to design better ways to communicate security information.

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 an unencrypted 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.

Acknowledgments

We are grateful to Yougu Yuan, Ed Feustel, Dan Wallach, Drew Dean, Mark Vilardo, and Carl Ellison for their helpful suggestions.

This work was supported in part by 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.

References

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

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

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

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

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

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

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

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

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

[10] The Mozilla Organization. `www.mozilla.org/download-mozilla.html`

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

[12] Bob Sullivan. "Scam artist copies PayPal Web site." *MSNBC.* July 21, 2000.

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

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

[15] 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.