Dartmouth Computer Science Technical Report 2002-430

Building Trusted Paths for Web Browsers

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Zishuang (Eileen) Ye

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 31, 2002

Examining Committee:

Sean Smith (chair)

Edward Feustel

Chris Hawblitzel

Carol Folt
Dean of Graduate Studies

Abstract

The communication between the Web browser and the human user is one component of the server-client channel. It is not the user but the browser that receives all server information and establishes the secure connection. The browser's user interface signals, such as SSL lock, *https* protocol header et al., indicate whether the browser-server communication at the current moment is secure or not. Those user interface signals indicating the security status of browser should be clearly and correctly understood by the user.

A survey of modern Web browsers shows the information provided by current browsers is insufficient for users to make trust judgment. Our Web spoofing work further proved that the browser status information is not reliable either.

We discuss the criteria for and how to build the trusted paths between a browser and a human user. We present an open source implementation of one of the designs–synchronized random dynamic (SRD) boundary, based on Modified Mozilla source code, together with its usability study results.

Acknowledgments

I want to express my heart-felt gratitude to Professor Sean Smith for his advisorship and unremitting support over the last one and half years. He trained me how to do research and how to write, encouraged me not to be intimidated by the difficult problems. He is so generous with his time and ideas. His intellectual creativity, perseverance and commitment would benefit me for the rest of my life. I could never thank him enough for being an excellent mentor and a wonderful friend.

My thanks also go to the other members of my thesis committee – Professor Edward Feustel and Chris Hawblitzel for the discussion during the course of the project, for the reading of the draft of this thesis and providing invaluable feedback.

I want to thank Densie Anthony and her assistant Robert Camilleri for designing the user study with me; Yougu Yuan for the cooperation in the Web spoofing project; Shan Jiang for letting me use his thesis format; Steve Hanna, Terry Hayer, Dan Wallach, Drew Dean, Jesse Ruderman, James Rome, Eric Renault, William Stearns for their help and suggestion. I also want to thank Meiyuan Zhao, John Marchesini,and other people in Computer Research Group and PKI lab. Their cooperation and friendship make the time in lab so enjoyable.

This work was supported in part by U.S. Department of Justice, contract 2000-DT-CX-K001. Therefore, my thanks go to them for making this project possible.

Finally, I am very thankful to Roger and my family back in China for their unconditional support and understanding.

Contents

List of Tables

List of Figures

Papers

1. E.Z. Ye, S.W. Smith “Trusted Paths of Web Browsers” accepted by 11th USENIX Security Symposium (Security ’02)

2. E.Z. Ye, Y. Yuan, S.W.Smith “Web Spoofing Revisited: SSL and beyond” Technical Report TR2002-417, Department of Computer Science, Dartmouth College

3. E. Ye, S.W. Smith, “Trusted Paths for Browsers: An Open-Source Solution to Web Spoofing” Technical Report TR2002-418, Department of Computer Science, Dartmouth College

Chapter 1

Introduction

In the last ten years, the World Wide Web has changed our life dramatically. Almost every aspect of social, financial, governmental activity has moved into Web settings. People use the Web for bank transactions, communication and entertainment. The Web has become so inseparable from our daily life that the security issues surround it have invoked concern.

1.1 Introduction of Web browsers

When people access the Web doing surfing, the activity involves three parts, the server, the Internet medium, and the client. However, the third part "client" actually contains a pair, a human user and a Web browser. It is usually ambiguous when we refer to a "client". Furthermore, there is common misunderstanding that the Web browser is the end of server-client communication. Computer scientists or engineers tend to think that as long as we reach the Web browser securely, we secure the whole server-client channel successfully. However, in the real world, the Web browser is the program that renders the Web pages, and the user is the person who

gets the information from the Web browser. They are not in one unit. The communication between these two is the one component of server-client communication. If we want to protect the whole server-client channel, we need to protect this component as well.

The Web browser establishes the connection to the Web server on the user's behalf, displays the Web pages and other information, such as the identity of the server and the connection status. The browser also occasionally collects the user's password for the local key store, as well as for server accounts. How can the user be informed of the server identity and connection status? Why should the user trust the server identity which is indicated by the browser, say "Trust.com" really is Trust.com rather than "Hacker.com"? Let's examine the browser features through which the user obtains this kind of information and makes his trust decision, using the Microsoft Internet Explorer (IE) and Netscape Navigator User Interfaces as models.

- **URL and location bar**

  At the top of the Web browser window, there is a location bar which displays the Web page source and the connection protocol. For example, a URL `http://www.dartmouth.edu/~pkilab/index.html` tells the client that the page source is `www.dartmouth.edu/~pkilab/index.html` and the connection protocol is *http*. If the connection protocol is indicated as *https*, the connection utilizes the *Secure Socket Layer (SSL)* technique.

- **Page info**

  The menubar is the toolbar with the "File" and "Edit" buttons. Clicking the "View" button on menubar gives a pop-up menu with an option "Page Info". The page info window gives more details about page source and server identity. However, most users do not bother checking it during the normal surfing activity.

- **SSL lock icon and URL**

  The SSL protocol uses encryption to protect the integrity and privacy of information during transmission. When a Web browser is in an SSL session, the URL in location bar would be displayed as

"https://...". The status bar, the bar at the bottom of the browser window details download process and help information, and shows a locked SSL lock icon. Clicking on the locked lock icon in some browsers or choosing the Page Info option would show the certificate of the Web site, which proves the server's identity.

1.2 Introduction of Problems

Above, we examined the browser features though which browsers tell the user which server they connect with and how they connect with. Is the information they provide sufficient for the user making trust decision? The answer is *no*. Let us see a non-exhaustive list of tricks that the attacker can play to fool the user.

- **RFC 1738 Flexibility**

  RFC 1738 specifies a Uniform Resource Locator (URL), the syntax and semantics of formalized information for location and access of resources via the Internet [24]. Users may base their decisions on the URL, because URL indicates where the page comes from. However, RFC 1738 permits more flexibility in host address expression than many people realize. The browser can display or accept a hostname as an IP address instead of the Web site name. It also permits the hostname portion include a username and password in addition to the machine name.

  As a case in point, in 1999, Gary D. Hoke [8, 16] created an HTML page which 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 board. 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 [2]: `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 which landed him

in legal trouble. However, browsers may present even more convincing disguises. As an example, versions of Netscape which we tested can accept IP address expressed as a decimal number (e.g., 3438189349) instead of the more familiar slot-dot notation. Since 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` where 3438189349 is the decimal IP address of the real server.

- **Typejacking**

  Tricky URLs can appear more standard, but contain incorrect hostnames deceptively similar to the hostnames users expect, then lure in unwitting customers. For example, `www.googl.com` also is a search engine site, which captures the careless google users.

  This technique can be used maliciously. As we recently saw in the headlines [17], 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. In many fonts, a lower-case "i" is indistinguishable from a lower-case "l", and many mail system enable the use click on a line in a email to establish the connection.

- **SSL servers**

  Although SSL is intended to protect information during transmission, SSL can not assure the server which is on the other side of transmission is the server the client intended to communicate with. For 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? Even if the answer is "yes", how is the user to know?

There are more tricks the attacker can play. Sometimes, until the attack becomes reality, the security hole would not attract enough attention. Although browsers have tried to provide information to enable the user

to make trust judgment, the tricks we mentioned above show the information is not sufficient. Why are the browsers willing to accept the weird looking URLs without providing a warning window? Why can Modus Media represent Palm Computing?

Even if the Web browsers display all the "right signals" (See following discussion.) to indicate that page A comes from "Trust.com" through an SSL session protected by a certificate registered by Trust.com, can the user really trust it? Are these browser features trustworthy?

The right signals include:

- **URL**: location bar displays as `https://www.trust.com/A.html`.
- **Page Info**: Page Info page says the file source is `www.trust.com/A.html`. It also shows the session's certificate is registered by Trust.com.
- **Status bar**: The locked SSL lock icon is displayed. Clicking on it, it may show the certificate registered by Trust.com.

Early in 1996, Felten et al [6] indicated that they were able to forge the location bar, Page info window and many other features of browsers. However, some researchers reported difficulty [13] in reproducing these results. It has been six years since Felten et al did their work, and Web techniques and browser user interface design have evolved since 1996. But can the new user interface resist the spoofing of the new techniques? In order to answer this question, we need to reexamine the browser features.

In Chapter 2 of this thesis, we present a survey of modern Web browsers on their security features, to further explore their insufficiency. In Chapter 3, we present our work on Web spoofing. Our Web spoofing work demonstrates that the user interface signals in modern browsers are not trustworthy, because they can be forged. In Chapter 4, we discuss the criteria of building trusted paths between the browser and the user. In Chapter 5, we present an open source implementation of our design, *Synchronized Random Dynamic (SRD)*

boundaries. In Chapter 6, we discusses the results of user study. Chapter 7 is the summary of the work done for this thesis. In Chapter 8, we discuss the future work. This thesis incorporates the material from our paper [27] *"Trusted Paths for Browsers"* and our technical report [25] *"Web Spoofing Revisited: SSL and beyond"*.

Chapter 2

Survey of Modern Browsers

2.1 Variety of Browsers

When we mention Web browsers, we immediately think about Internet Explorer (a Microsoft product) and Netscape Navigator (now owned by AOL). Both IE and Netscape have a series of versions with improved features and implementations. These two families are the most popular ones in Web browser domain. However, more Web browsers exist than one may realize. According to *Dan's Web tips* [21], more than a hundred Web browsers exist. Some of them are for special use, like *Amaya*, which mainly functions as a testing bed for W3C new technology; some are favorites because of their simplicity and speed, like *lynx*, a Linux Web browser that only displays text; some use 3D visual effects, like *Browse3D*, a browser that displays Web pages on the walls of a 3D room, so the user can read five pages at the same time; some are written in other languages instead of C/C++, like *HotJava*, a Web browser written in Java.

Although there are so many types of Web browsers, there are few of them that can compete with the big two, IE and Netscape. Few of them run on multiple platforms and support continually updated Web techniques.

Opera [10] is an exception. The Opera company was founded in 1995 in Norway. However, its products did not win world-wide attention until 2000. It is touted as the fastest and safest Web browser on earth. When Opera 4.0 for Windows appeared in early 2000, more than 1 million copies of the Opera browsers were downloaded during the first month [11]. Opera version 5.0 for Windows, the free ad-sponsored browser, was launched in December 2000, and more than 2 million users from all over the world downloaded the browser during the first month. In the same month, Opera signed strategic agreements with IBM, AMD, Ericsson, Psion, Qualcomm, PalmPalm and Screen Media. Now Opera is released for Windows, Linux/Solaris, Mac OS, OS/2, Symbian OS, BeOS and QNX. It is difficult to count how many Opera browsers are used, because Opera can claim itself to be IE, Netscape, or Mozilla when its identity is asked by the server.

In 1995, Netscape Inc. released the source code of Netscape to the open source community. The Mozilla organization was then founded intending to continue the Netscape project as an open source project. Most of the Mozilla source are maintained by Netscape (AOL) engineers. However, when Netscape (AOL) releases its product, it does its own QA test. Netscape (AOL) put much effort in its own user interface design and the last fine tuning step. This is why although Mozilla has same core structure as Netscape, it may perform differently.

Konqueror [20] is a Web browser and also the file manager of KDE desktop environment, only for Unix workstations. However, because it supports quite advanced Web techniques and seamlessly inter-operates with other KDE applications, it commands great attention in open source community.

2.2 Testing

We chose several mainstream Web browsers as examples of modern Web browsers and tested their responses to malicious URLs and specially designed Web pages. The browsers are IE 5.0 for Windows, Netscape 6.0 for Linux, Netscape 4.76 for Linux, Opera 5.0 for Linux, Mozilla 0.9 for Linux and Konqueror.

The malicious URLs are

```
http://www.cnn.com:mainpage@2175456613/~sws/0/
http://www.cnn.com:mainpage@129.170.213.101/~sws/0/
```

21753456613 is the digital expression of `www.cs.dartmouth.edu`; 129.170.213.101 is the IP address. The true page is `http://www.cs.dartmouth.edu/~sws/0/`. However, the URLs mislead the client to believe it actually comes from CNN with "mainpage" as login name and the following part as password. We examined whether a browser would interpret the strange looking URL successfully.

It is natural to think of "an SSL session with Server A", "a non-SSL session from Server B". However, the Web page source is not necessarily limited to be one server. The elements in the page may come in SSL session or non-SSL session from different servers. It is interesting to see the browser responses to these specially constructed pages.

The specially designed Web pages are:

- A page from Server A in an SSL session, as shown in Figure 2.1.
- 1. A page from Server A in a non-SSL session contains four frames. One frame comes from Server A in a non-SSL session; one frame comes from Server B in an SSL session; two others are blank: a non-SSL page from A has an SSL frame from B, as shown in Figure 2.2:

  2. A page from Server A in a non-SSL session contains an image from Server B in an SSL session: a non-SSL page from A has an SSL image from B.
- 1. A page from Server A in an SSL session has four frames. One frame comes from Server A in an SSL session; one frame comes from Server B in a non-SSL session, the other two frames are blank: an SSL page from A contains a non-SSL frame from B, as shown in Figure 2.3.

  2. A page from Server A in an SSL session contains an image from Server B in a non-SSL session: an

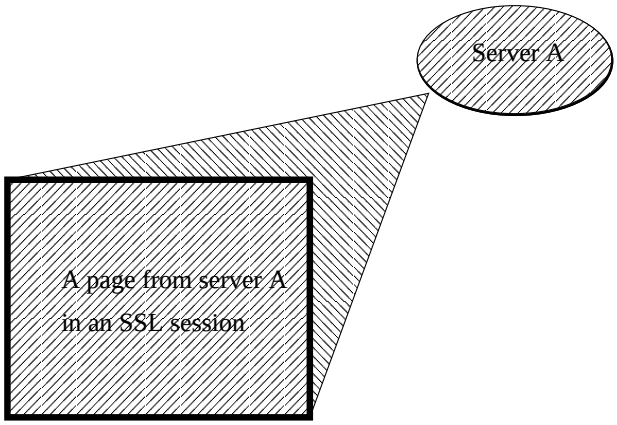


Figure 2.1: A single page from server A in an SSL session.

SSL page from A contains a non-SSL image from B.

- 1. A page from Server A in a non-SSL session which contains four frames. One frame comes from Server A in an SSL session; one frame from Server B in an SSL session; the other two frames are blank: an SSL page from A has an SSL frame from B, as shown in Figure 2.4.

  2. A page from server A in an SSL session contains an image from server B in an-SSL session: an SSL page from A has an SSL image from B.

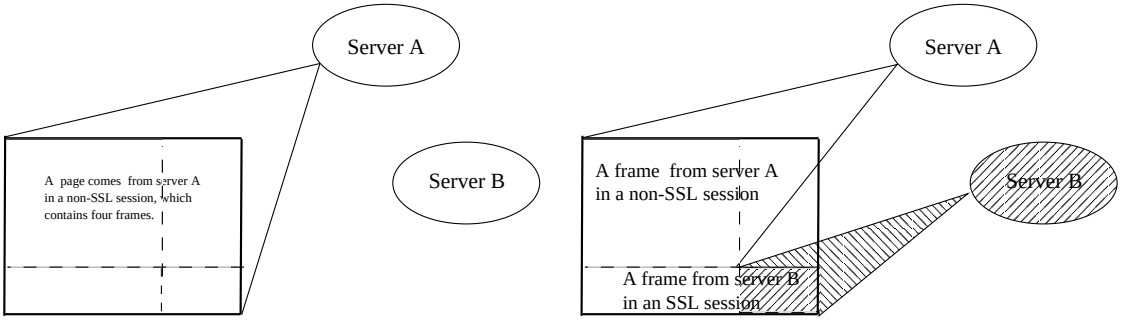
We collected the information about the SSL warning window and the server certificate window.

We list the test results in tables Table 2.1, Table 2.2, Table 2.4, Table 2.5.

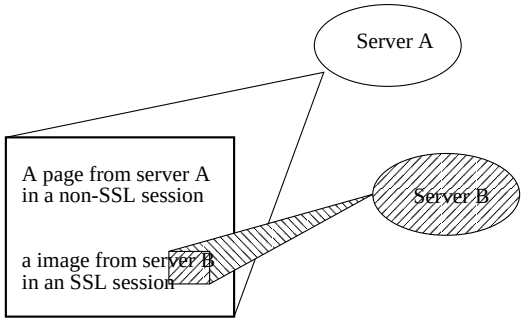
2.3 Summary

From the results in above referenced tables, we can see that Web browsers have inconsistent responses to malicious different testing pages.

Opera warns the client about the strange looking URLs and response to non-SSL session elements inside a

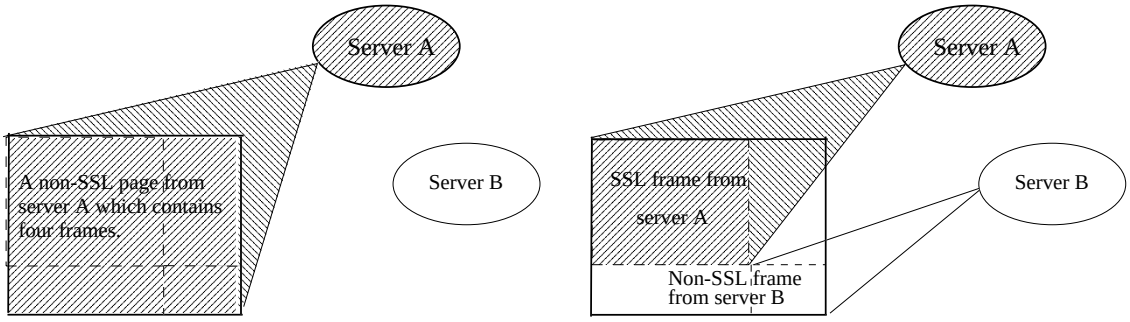


A page from server A in a non-SSL session which contains four frames.
One frame is from server A in a non-SSL session. One frame
is from server B in an SSL session. The others are blank.

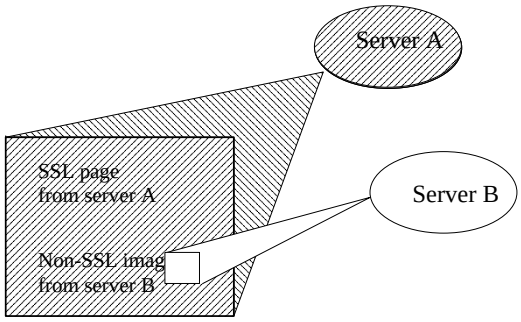


A page comes from server A in a non-SSL session
which contains an imagine from server B in an SSL session.

Figure 2.2: A page from Server A in a non-SSL session with an SSL element from Server B

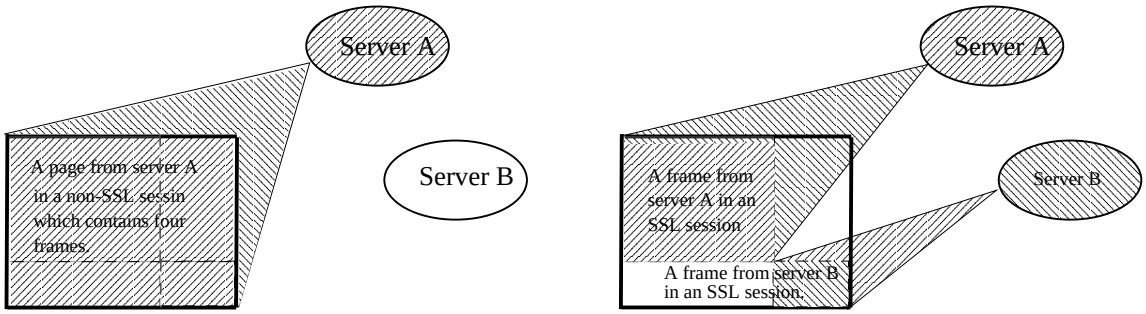


An SSL page from server A which conatin four frames with different sources.
One frame from server A in an SSL session, one frame from server B in an non-SSL session.
Two other frames are blank.

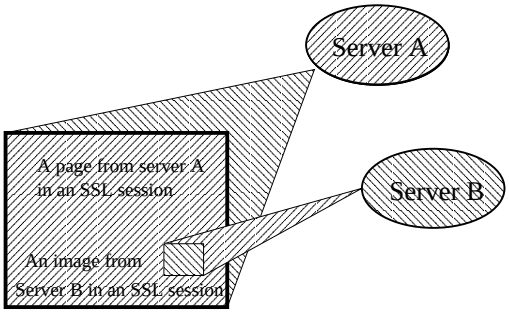


An SSL session page from server A which contain a non-SSL image from server B.

Figure 2.3: A page from server A in an SSL session which contains a non-SSL element from Server B.



A page from server A in a SSL session which contains four frames. One comes from server A in an SSL session. One comes from server B in an SSL session. The other two are blank.



A page from server A in an SSL session which contains an image from server B in an SSL session.

Figure 2.4: A page from server A in an SSL session which contains an SSL element from Server B.

Browser Names	Link 1	Link 2
	http://www.cnn.com:mainpage @2175456613/∼sws/0/	http://www.cnn.com:mainpage @129.170.213.101/∼sws/0/
opera 5.0 for Linux	both links invoke security warning about the strange looking of URL, the user can choose continue or not	security warning
Konqueror	parse through, the page is displayed; the URL automatically change to be http://www.cnn.com @2175456613/∼sws/0/	parse through, the page is displayed; the URL automatically change to be http://www.cnn.com @129.170.213.101/∼sws/0/
Netscape 6.0	parse through, the page is displayed; the URL does not change	parse through, the page is displayed; the URL does not change
Mozilla 0.90 for Linux	parse through, the page is displays; the URL does not change	parse through, the page is displays; the URL does not change
Netscape 4.76	parse through, the page is displayed; the URL automatically change to be http://www.cnn.com @2175456613/∼sws/0/	parse through, the page is displayed; the URL automatically change to be http://www.cnn.com @129.170.213.101/∼sws/0/
IE 5.0	parse through, the page is displayed; the URL doesn't change	parse through, the page is displayed; the URL doesn't change

Table 2.1: Different behavior of different browsers toward special links.

Browser	SSL page from server A
opera 5.0 for Linux	No warning window for entering or leaving SSL session; a big lock icon shows on tool bar with green secure level indication; clicking on the security icon can invoke the security configuration, but no certificate information shows directly; install server certificate in the first time communicating with server
Konqueror (KDE 2.0)	Warning windows for entering or leaving SSL session; a big security lock icon shows on tool bar; after entering SSL session, the lock locks; clicking on it, the certificate information shows directly
Netscape 6.0 for Linux	warning windows for entering or leaving SSL session; a small lock icon in status bar indicate the SSL session; the server certificate shows directly by clicking the lock icon
Mozilla 0.90 for Linux	warnings for entering or leaving SSL session; a small lock icon in status bar to indicate SSL session establishing; clicking security icon has no reaction; Page Info display the server certificate directly
Netscape 4.76 for Linux	warning for entering or leaving SSL session; a locked icon shows on both tool bar and status bar indicating the SSL established; clicking on them, the security configuration page popped up which contains a link to display certificate
IE 5.0	warning window for entering SSL session; a small locked icon shows on status bar after that; clicking the lock icon, server certificate shows directly; the server installs certificate in the first time communication

Table 2.2: Different behavior of different browsers during a single SSL session

Browser	Non SSL page from Server A with one SSL frame from Server B	Non SSL page from Server A with one SSL image from Server B
opera 5.0 for Linux	no warning for entering or leaving SSL session; the security lock icon keeps open; no certificate information about server B displays; the log in server B indicates the success of file delivery; server B installs its certificate in the first time communicating	no difference from the frame page
Konqueror (KDE 2.0)	no warning for SSL element in page; no SSL connection indication; the security lock icon keeps open; no certificate information of server B displays; the log in server B shows the SSL element has been delivered	no difference from the frame page
Netscape 6.0 for Linux	warning window for entering or leaving SSL session; server B installs certificate; a blur locked lock shows on status bar to indicate partial SSL session; clicking on it, says server A does not support certificate, no certificate of server B displays; the log in server B indicate the files has been delivered through SSL session	no warning for entering SSL or leaving session; the security icon keep unlocked; clicking on it, it says server A does not support encryption; server B install certificate in the first time communicating; the log in server B shows the image file is delivered
Mozilla 0.90 for Linux	warning for entering a weak SSL session; warning for insecure element in page; security icon keeps unlocked after that, no reaction by clicking it; the log in server B shows the file has been delivered; server B installs certificate at the first time communication	no warning for SSL element in page, no other difference from the frame page;
Netscape 4.76 for Linux	warning for entering SSL or leaving session; the security icons keep unlocked; server B installs its certificate in the first time communication; clicking on the lock icons brings a page with no link for showing the certificate of server B	no difference from the frame page
IE 5.0	no warning for SSL session; lock icon keeps unlocked; server B installs its certificate in the first time communicating; the log in server B shows the SSL item is delivered	no difference from the frame page

Table 2.3: Different behavior of browsers toward a non-SSL page from Server A which contains one SSL element from Server B

Browser	An SSL page from Server A with one Non SSL frame from Server B	An SSL page from Server A with one Non SSL image from Server B
opera 5.0 for Linux	no warning window for non SSL element in page; the security lock icon keeps open, means no SSL session; the URL shows as https; the log in server B shows the success of file delivery through non SSL session; server A installs certificate in the first time communicating	no difference from the frame page
Konqueror (KDE 2.0)	warning window for entering SSL session; no warning for non SSL element in page; clicking on the locked security icon, the certificate of server A displays; there is a warning for leaving SSL session when the browser fetches the non SSL element during loading, but lock icon keep locked after that; the log in server B shows the insecure file is delivered through non SSL session	no difference from the frame page
Netscape 6.0 for Linux	warnings for entering or leaving SSL session; a warning for the insecure non SSL element in page; a blur locked security icon shows on status bar after that; clicking on it says server A does not support encryption; URL shows as https; the log in server B shows the file has been delivered through non SSL session	no difference from the frame page
Mozilla 0.90 for Linux	warning for establishing a weak SSL session; warning for the insecure element in page, lock icon locked; warning for establishing a weak SSL session again; lock icon change to be unlocked after that, no certificate popup by clicking on it; URL shows as https; Page Info display certificate; warning for leaving SSL session	warning for insecure element in page when loading page, no other difference from the frame page
Netscape 4.76 for Linux	warning for entering SSL session; no warning for non SSL element in page; server A installs certificate in the first time communication; security icons keep unlocked; clicking on it brings a page with a link to show the certificate of server A; the log in server B shows the file has been delivered through non SSL session	warning for entering SSL session; warning for non SSL element in page; server A install certificate in the first time communication; security icons lock after that; the log in server B indicate that the page has not been delivered
IE 5.0	warning window for entering SSL session; warning window for insecure non SSL element in page; IE gives out three non SSL element warnings for one non SSL frame and two blank page frames; no lock icon is showed up	warning for entering or leaving SSL session; asking for user interaction to display insecure element or not; no matter chose display or not, the log in server B shows the insecure element has been delivered to client; security lock icon shows up on status bar after that; clicking on it, display server A certificate

Table 2.4: Different behavior of browsers toward an SSL Page from Server A which contains one non SSL element from Server B

Browser	An SSL page from Server A with one SSL frame from Server B	An SSL page from Server A with one SSL image from Server
opera 5.0 for Linux	no warning for two SSL sessions; the security lock icon locks, with indication of low security ; no certificate information shows directly after clicking the security icon; the log in server B shows the file delivery through SSL session; the page fails displaying in the first time, but shows after clicking Back and Forward button; both server A and B install certificate in the first time communication	no difference from the frame page
Konqueror (KDE 2.0)	no warning for two SSL sessions; one waring for entering or leaving SSL session; the security lock icon locks after that; clicking on it shows the certificate of server A; no SSL connection to server B shows; the log in server B shows the file has been delivered through SSL connection	no difference from the frame page
Netscape 6.0 for Linux	warnings for entering or leaving SSL session; both server A and server B install certificates in the first time communication; a locked icon shows on status bar after that; clicking on it, the certificate of server A shows up directly; no information about the second SSL session shows	no difference from the frame page
Mozilla 0.90 for Linux	two warnings for entering two SSL sessions; the security icon locks after that; both servers install certificates at the first time communication; Page Info only displays server A certificates; warning for leaving SSL session	no difference from the frame page
Netscape 4.76 for Linux	One warning for entering or leaving SSL session; security icons lock after that; clicking on them bring a page which has a link to show the certificate of server A, no link for the certificate of server B; both servers install certificates in the first time of communication; the log in server B indicate the file has been delivered through SSL session	no difference from the frame page
IE 5.0	warning for entering or leaving page; warning for two non SSL blank frames; no security lock icon shows; the log in server B shows the file has been delivered	without frames, warning for entering or leaving SSL session; the small security icon shows on status bar, display certificate of server A by clicking on it, no link for displaying the certificate from server B

Table 2.5: Different behavior of browsers toward an SSL page from Server A which contains one SSL element from Server B

SSL session, however, it doesn't provide certificate information directly. That means the user can not check the certificate expiration date and the finger print. Opera does not warn the user if the server's certificate has expired. However, Opera warns the user if the server name does not match the server name on the certificate. Opera has no warnings on entering or leaving SSL also.

All other browsers do not warn the client about the strange looking URLs, but they all provide certificate information directly. IE 5.0, Netscape 6 and Mozilla 0.9 all warn the client if there is a non-SSL element in SSL session, while Konqueror and Netscape 4.76 do not. Versions of Netscape and Mozilla warn users about the SSL elements inside a non-SSL session, while the others do not. No warning of SSL session elements inside a non-SSL session may enable the attacker launch a *denial of service attack* to server B. All the browsers except Opera accept the multiple SSL session, although they can not display multiple certificates.

There are no standards for how the browsers should response to different situations. The responses depend on their internal structure, how they interpret URLs and how they implement an SSL session. For example, Netscape 4.76 and Netscape 6 have total different internal structures, because Netscape 6 was rebuilt from scratch. They were expected have different behaviors. However, the lack of standards brings confusion to the client. Which behavior means what? Does the lock icon locked mean safety? What does a blur lock icon mean? Obviously, we need a more consistent and clearer understanding of the Web browser by the user.

Chapter 3

Web spoofing

In this chapter, we will examine the reliability of the information provided by the Web browser. For example, when the browser displays a certificate of server A, can we believe it really is server A at the other end of the communication?

Here we introduce a working definition of Web spoofing [25]: the malicious action causing the reality of the browsing session to be significantly different from the mental model a reasonably sophisticated user has of that session.

In a typical Web spoofing attack, the attacker presents the user carefully designed Web pages, and tricks the user into believing that he is communicating with the server he intends to. Web spoofing is possible because all the information which the server provides to the user is collected and displayed by the browser instead of being received by the user directly. Even in SSL sessions, the user judges whether a connection is secure based on information delivered through the browser like the SSL icon, warning windows, location information, and certificates. If the attacker is able to manipulate the browser user interface, he is able to take advantage of the user mental model, and lead the user into making a wrong trust judgment.

3.1 Previous Work

Princeton In 1996, Felten et al at Princeton [6] originated the term *Web spoofing* and explored spoofing attacks 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. They used a real SSL session started from the attacker server to spoof SSL. Source code is not available. According to the paper, the attack used JavaScript to rewrite the hyperlink information shown on the real 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). Although the paper mentioned that, in 1996, it was possible to spoof the function of menu items, like viewing document source, how to reproduce this today is challenging, since menu items are now located in pop-up menus. Furthermore, using a genuine SSL session exposes the attacker to the certification process.

UCSB In 1997, De Paoli et al at UCSB [13] tried to repeat the Princeton attack, but could not 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 HTML document that embeds a *spy applet* with its stop method overridden (The Thread.stop method has been deprecated by Sun [18] years ago, but it does not necessarily mean this method can not be used by hacker any longer.). 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 Web site, then displays an impostor page for that site in order to steal sensitive information, such as a

credit card number.

CMU In related work in 1996, Tygar and Whitten from CMU [22] 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* a SSL connection, when no SSL connection exists. We did it.

3.2 Web Spoofing Experiment

Web techniques and browser user interface both evolved a lot since Felten et al. did their work. Are the modern Web browsers still vulnerable to Web spoofing? Is it possible spoof a SSL session?

Our experimental results show: for standard browser configuration, every interface including the existence of SSL session and the alleged certificates can be forged.

Target We target WebBlitz, a Web version of our campus e-mail system, hosted by *basement.dartmouth.edu*. WebBlitz is a nice target because our local community has immediate familiarity with it, it is easy to test human reaction when people confront the spoofing effect. WebBlitz uses an SSL session. It also involves sensitive data: its login name and password are valid for many other academic transactions. So a successful spoof against WebBlitz would permit exploration of many interesting features.

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 easily 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 the new browsers.

3.3 Implementation Details

3.3.1 Initial Attempt

The Princeton paper seemed to imply that a malicious server could change or overwrite the client's location bar. We tried to do this, and failed.

3.3.2 A New Window

We then tried to open a new window with only the location bar turned off and other bars on.[1]

However, this approach creates problems in modern browsers:

- With some Netscape Navigator configurations, turning off only the location bar invokes a security alarm.
- With Internet Explorer, we can turn off the bar without triggering alarms. But unfortunately, we cannot convincingly insert a fake location bar, since a noticeable empty space separates browser bars (where the location bar should be) and server-provided content (where our fake bar is).

Leaving *all* the bars in the new window makes spoofing impossible, and turning off *some* caused problems.

[1] Subsequently, we learned that this is what the Princeton work did. [3]

However, we can turn off *all* the bars in the new window, and open an empty window, which cause no problems. In this case, we are able to replace all of bars with our own fake ones—and they all appear with the correct spacing, since the whole window is under our control.

We create fake bars using images, culled from real browsers via *xv*. Since the browser happily gives its identity to the server, we know whether to provide Netscape or Internet Explorer images. Our first attempts to place the images on the empty 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 until fill up the whole window; we address that problem by constraining the frame to exactly the size of the bars image.

3.3.3 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 have a highlighted 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 3.1 and Figure 3.2 show samples of our fake tool bar interaction and a real tool bar interaction, for Netscape 4.75.

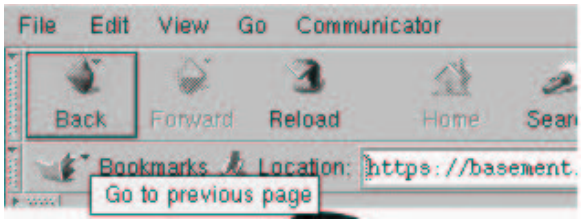


Figure 3.1: Sample fake tool bar pop-up in Netscape.

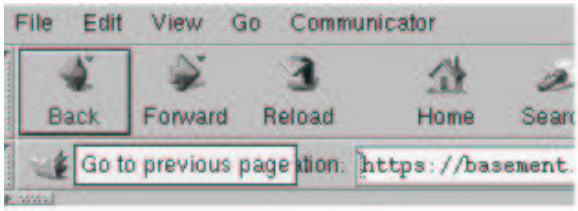


Figure 3.2: Sample true tool bar pop-up in Netscape.

3.3.4 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 4.76, 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, to get convincing fake bars in Netscape window, we needed to load them as backgrounds in frames. This technique creates a problem for spoofing pop-up menus: the pop-up layer is constrained to the width of the frame, and the width is too small for convincing pop-up menus. However, this problem has a simple solution: we replace a multi-frame approach with one frame—containing a merged background images.

3.3.5 SSL Icons and the Status Bar

To convince the client that a secure connection has been established, we need the locked lock icon displayed on the status bar.

In our attack, our window displays a fake status bar which is completely under our control. When we wish the user to think that an SSL session is underway, we simply display a fake status bar containing a lock icon. 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. Our approach enables the adversary to fool even a user who has configured their browser to accept only a specialized trust root.

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 successful 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.

3.3.6 SSL Warning Windows

Browsers typically pop up warning windows when users entering or leaving an SSL session. 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.

  This fake warning window would not display properly if it covers genuine input fields or buttons. There are two ways to avoid this phenomenon happen:

  - Replacing the input fields or buttons with images. Then the warning layer is just showed on top of another image.
  - Carefully adjust the position of the warning layer, so it would not cover input fields or buttons.

  Our experience show the second approach is better than the first one, because to some users, the redraw of the background image is noticeable.

- For Internet Explorer, we use a Modal Dialog with HTML that shows the same content as the warning window. Unfortunately, there is a "Web page dialog" string appended on the title bar in the spoofed window, which may be noticeable to careful users.

Figure 3.3 and Figure 3.4 show the fake and true warning windows for Netscape 4.75.

3.3.7 SSL Certificate Information

The SSL protocol enables the client and the server to authenticate themselves to each other. But 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 of the server. When establishing an SSL session with a server, the browser evaluates the server certificate; if all the information contained in the certificate is correct,

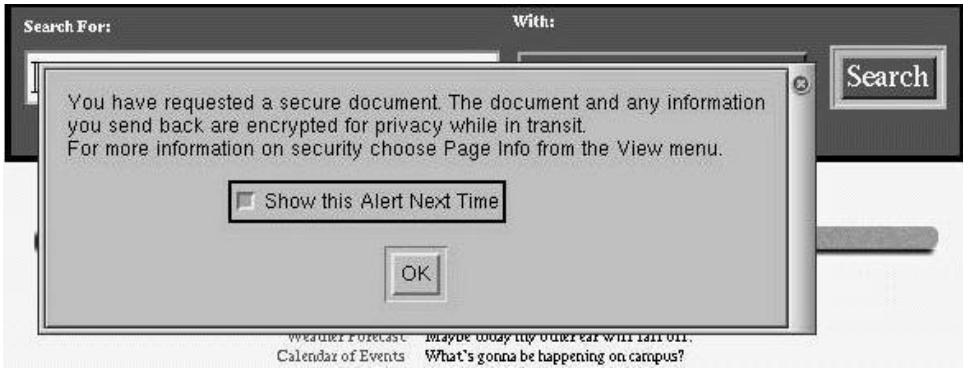


Figure 3.3: Our fake SSL warning window in Netscape.

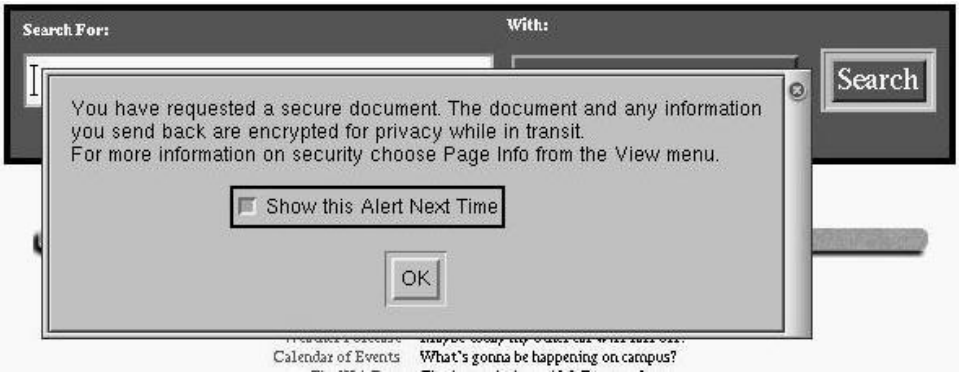


Figure 3.4: A true SSL warning window in Netscape.

the browser then establishes SSL session keys sharable only by the entity who knows the private key matching the public key in the certificate.

A locked SSL lock icon would be launched to indicate that, in the current session, traffic between the browser and server is encrypted and protected against eavesdropping and tampering.

However, a sophisticated user will not only keep an eye on the SSL icon, but will also inspect the server certificate information to evaluate whether or not a "trusted" session exists. 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 also can do 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 did not implement that case.

3.3.8 Location Bar Interaction

Editable location bars is one aspect of spoofing that has been made more difficult by evolving Web technology. Here, we need to either gamble on the user's configuration, or give up on this behavior.

As we noted, JavaScript cannot change a real location bar, but it can hide the real location bar, and put a fake one in the position where it's expected. The fake location bar can show the URL that the client is expecting to see. But besides displaying information, a real location bar has two common interactive behaviors: receiving

input from the keyboard, and displaying a pulldown history menu.

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

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

3.3.9 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.

3.3.10 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 function.

The hyperlink to our Web spoofing site is

```
< A href= ‘‘realsite-address’’ onclick=‘‘return openWin()’’ >
```

Bringing the mouse over this link displays “realsite-address” on the status line.

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

3.3.11 Pre-Caching

In order to make the browser looks real, we use true browser images grabbed from screen.

Since our spoof uses a fair number of images, the spoofed page might take an unusually long time to load on a slow network. This is not desirable from an attacker's point of view. To alleviate this problem, we use pre-caching: each page loades 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.

3.4 Extensions and Discussion

Collecting Browsers Information A critical part of making our spoof convincing is tuning the content for the user's browser. Is it Netscape or IE and which version? The more the attacker knows about the user's browser, the more convincing the attacker can do.

There are quite a bit of information that the server can know about the browser, such as the screen size and the window size. We can change the pop-up window size according to this information. BrowserSpy [23] can estimate the bandwidth the user using by rendering a 50k page and testing the loading time. This technique is desirable for the attacker, because Web spoofing may use a fair amount of images, video or audio data.

There is some information only exposed to signed scripts. For example, signed JavaScript can ask for permission to access the *history* object which may enable the attack forge the history link inside the browser user interface. Signed script also may access the *preference* object, which can tell the attacker whether the image switch is trued on or not, what security level the browser is running, which theme the user chose. We did not allow signed script in our Web spoofing. If we relax this restriction, more interesting scenarios may arise. One can imagine scenarios where the query is embedded in an "innocent" signed script. Because the script is signed by its programmer, it can access the preference information. The information collected by this "innocent" script can be sent back and used for a later attack.

Minimum Exposure A cautious adversary might worry that the more one cheats, the more likely one will be found out. To minimize the exposure, we limit the clients to those using common browser/OS configurations that we can handle; we send the rest to the real site. Some other cautionary tricks that could be applied include:

- After using CGI or any other program to collect the sensitive data, we could redirect the client 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 looks like a running time error that crashes the browser, which still is a common behavior of modern browsers.

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 also explored Netscape 6 after that.

When we set up the experimental attack for these two families, we have fairly different experiences. Some techniques supported by one browser may not work in another. However, 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 *layer*, because it is not a *W3C* standard, but the same functionality can be provided by *DIV* feature.

- **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 how 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.

3.5 Conclusions for Web Spoofing

What we are able to do To summarize our experiment: for Netsape 4.75/4.76 on Linux and Internet Explorer 5.5 on Windows 98, using unsigned JavaScript and DHTML:

- We can produce an entry link that point to the target Site S.
- If the user clicks on this link, if his browser/OS combination we do not support or his browser does not have JavaScript enabled, we send them really will go to Site S.
- Otherwise, their browser opens a new window that appears to be a functional browser window, presents the appearance of the target site. Signals, bars, location information, and much browser functionality all appear correct in that window. Except, the user is not visiting that site at all; he is visiting ours. The bars, buttons, and other functional parts actually are images linked with JavaScript.

- Furthermore, if the user clicks on a "secure" link from this new window in order to try to start an SSL session, we can make convincing entering SSL warning pop-up, then display the SSL lock icon. If the user clicks on the buttons for security information, he or she will see the expected SSL certificate information. All appear as the user expects—except there is no SSL connection exists. All the sensitive information which the user enter in the input fields, like password and login name, is being sent in plaintext to us.

Limitions of Our Web Spoofing 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 bar appears to depend on the user's font preferences, which we can not learn.
- We cannot yet obtain the user's history information to implement the pulldown history options.
- As to Netscape 6, we can not acquire which theme the user uses, classic or modern. Netscape 6 can have classic Netscape appearance like Netscape 4.7, or have a new modern appearance. The use can chose either of them. This information is kept in *preference* object. Only signed script with granted privilege can access it. In our case, we can not know which theme the user chose.

Implications What are the risks to Web users that our experiment indicated? We do not think a convincing long-lived spoofed Web are likely, because first, until now it still is difficult to spoof every browser function; second, it is impossible to handle every platform. However, short-lived sessions with narrow user behavior are much more susceptible. In theory, we can send our friend a link to our spoof Web, capture his login name and password.

3.6 Countermeasures

How can users protect themselves from Web spoofing?

Short-term Solutions

- **Disable JavaScript**

  Web spoofing techniques depend mostly on JavaScript. If the user disable browsers JavaScript, he will deny this attack. However, modern Web pages rely on JavaScript so much that many feel disabling it is impractical for general Web surfing.

- **Customization**

  Tygar and Whitten suggested customization as a countermeasure against Trojan Horse applets. Customization of browsers setting is also an effective way to enable users to detect Web spoofing. Although unsigned JavaScript can detect the platform and browser which the client is using, it can not detect the detailed window setting which may affect the browser display. The browser Opera has more customizable interface than other browsers. From this point of view, Opera is more secure than other browsers.

- **Disable pop-up windows**

  Disabling pop-up window can stop Web spoofing from opening a new window completely controlled by attacker. Unfortunately, disable pop-up only implemented as an option in browser Konqueror, which comes with KDE 2.0, only for Linux. Furthermore, using pop-up window for advertisement has become routine, which generate significant revenue for many e-commence Web site. Disabling pop-up new window may be too restrictive.

Long-term solutions. 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.

None of above solutions are strong enough to be a general solution for preventing Web spoofing. A ideal browser should be a platform which can enable all the modern Web techniques to be full functional, and at the same time supply unspoofable features to indicate the communication security.

In next chapters, we will discuss how to design an unspoofable channel for the browser-user communication and provide the implementation to one of these solutions.

Chapter 4

Designing Trusted Paths between Browser and User

From our experiment results in Chapter 2 and Chapter 3, we conclude that modern browsers do not provide sufficient or trustworthy information. Therefore, the security judgment made by the user based on this information is questionable. As people more and more rely on the Web for information, Web browsers have become the main stages for e-commerce, personal communication and all kinds of issues. How to effectively and correctly deliver the security information to the user is a challenge to the browser user interface design. A pleasant appearance should only be one of the criteria. Security should also be kept in mind.

Before we move on to the discussion of the criteria, let us try to formalize the problem.

4.1 Basic Framework

We provide a slightly simplified model.

Web page C is delivered to the browser from server A through an SSL channel or a non-SSL channel. The browser display the page content as a set of graphical element: *content*(C). The browser also displays a set of graphical elements to indicate how this page C comes, through SSL or non-SSL, server name A et al. This information is formalized as *status*(C). The status and content combine together, are presented in the window

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

The users perform function *recog* to get the status information out of the window information and perform function *eval* to evaluate the status, then make a trust decision by performing function *judge*:

- They can recognize the material that belongs to status information:

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

- They can decide to trust the content or not based on an evaluation of that status information:

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

Web spoofing attacks can work because there is no clear difference between *status* and *content.* They may have collisions: there exist pages C_1, C_2 such that *content*$(C_2) \cap$ *status*(C_1) can be substantial. This overlap permits a malicious server to provide content C_2 which tricks users into recognizing it as status from C_1:

$$\textit{recog}(\textit{window}(C_2)) = \textit{recog}(\textit{status}(C_2) \cup \textit{content}(C_2))$$

if *status*(C_2) is empty,

$$\textit{recog}(\textit{window}(C_2)) = \textit{recog}(\textit{content}(C_2))$$

If *content*(C_2) = *window*(C_1) = *status*$(C_1) \cup$ *content*(C_1)

$$\begin{aligned}
\textit{judge}(\textit{window}(C_2)) &= \textit{eval}(\textit{recog}(\textit{window}(C_2)) \\
&= \textit{eval}(\textit{recog}(\textit{content}(C_2))) \\
&= \textit{eval}(\textit{recog}(\textit{status}(C_1) \cup \textit{content}(C_1)) \\
&= \textit{eval}(\textit{status}(C_1))
\end{aligned}$$

Consequently, users mistakenly judge C_2 as they would judge C_1.

4.2 Trusted Path

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

- making clear differences between *status* and *content*. Therefore, *content* can not impersonate *status*.
- making it impossible for *status*(C) to be empty, so browsers always is able to tell the user that *status*(C_2) exists.

4.3 Design Criteria

What are the criteria that a trusted path solution must satisfy?

At first, the trusted paths 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 3.5 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.

Second, the trusted paths 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.

4.4 Prior Work

As discussed in section 3.6, Tygar and Whitten [22] suggested that using customization could defend hostile Java applet spoofing. Customization also can be used to defend general Web spoofing. However, this approach requires the user know how to customize their Web browser which violate the minimizing user work criterion.

Compartmented Mode Workstations (CMW) is a secure operating system developed by Department of Defense and National Security Agency for multilevel classified work in the middle 80s. CMW is rated as B1 in the NSA orange book with many features of the B2 and A levels. All files in CMW are ranked according to their security level, classified, confidential or public. All windows in CMW are color labeled at their top to indicate their security level. Read-down/write-up policy is enforced by window manager. All activities can be audited [14].

CMW introduced the classic multi-level security model into OS design. However, a Web browser running on top of CMW won't be a solution for Web spoofing. Because 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 4.5.2.

4.5 Considered Approaches

In section 3.6, we discussed some short-term solutions for defending Web spoofing. Each of them has certain limitions which prohibit them being an effective long-term one. Those limitions are what we should avoid when we design our long-term solution.

4.5.1 *status* not empty

One way to defend against Web spoofing is making it impossible for *status* to be empty. One possible approach is prevent the bars or buttons from being turned off in any window. This approach would overly constrict the display of server pages. A concrete example is Opera 5.0 for Linux: the "Forward" and "Back" buttons with a flashing advertising bar can not be turned off. All the server content only can be displayed within the right-low area, which is very restrictive. Furthermore, this approach still will not cover a broad enough range of browser-user channels. For example, the attacker still can use images to spoof pop-up warning windows. If the server can send images, they can forge a nested window with its own status information.

4.5.2 Distinguishing *status* and *content*

The other way to defend Web spoofing is clearly label the *status* information, in order to make it distinguishable from *content*. We considered several approaches before we reached our final design.

MAC phrase One approach would have the user enter an arbitrary phrase, as "MAC phrase", at the start-up time of browser. The browser could then insert this MAC phrase into each *status* window, including certificate status window, SSL warning boxes etc., to authenticate it. However, we think that this approach may require too much work from the user.

Meta-data title We considered having some meta-data (like page URL) displayed on the window title. The title information is sent by Mozilla to machine window system, so we can enforce the true URL always is displayed on the window title. 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.

Meta-data windows We considered having an extra window always open, which would have a unique label to prove its genuineness, and display page meta-data, such as URL, server certificate, etc.

We felt that this approach would not be effective. The first reason is separating the data from the content window would make it easy for users to ignore the meta-data. The second reason is the user may not understand the text material even they notice the meta-data.

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. Colors or images are generally more effective than 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 the server 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, like how we spoofed SSL warning window.

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

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 separation would appear to fail the effectiveness and user-work criteria:

- The security level information appears later, and in a different part of the screen.
- The user must explicitly click on the window to get it to focus, and *then* confirm the status information.

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

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

4.6 Synchronized Random Dynamic Boundaries

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

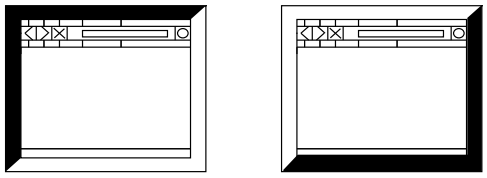


Figure 4.1: *Inset* and *outset* border styles.

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.

4.6.1 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 4.1). At random intervals (several seconds), the browser would change the styles on all its windows. Figure 4.2 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.

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.

4.6.2 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.

	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*
***S**RD*	***Y**es*	***Y**es*	***Y**es*	***Y**es*

Table 4.1: Comparison of strategies against design criteria.

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 5.8.

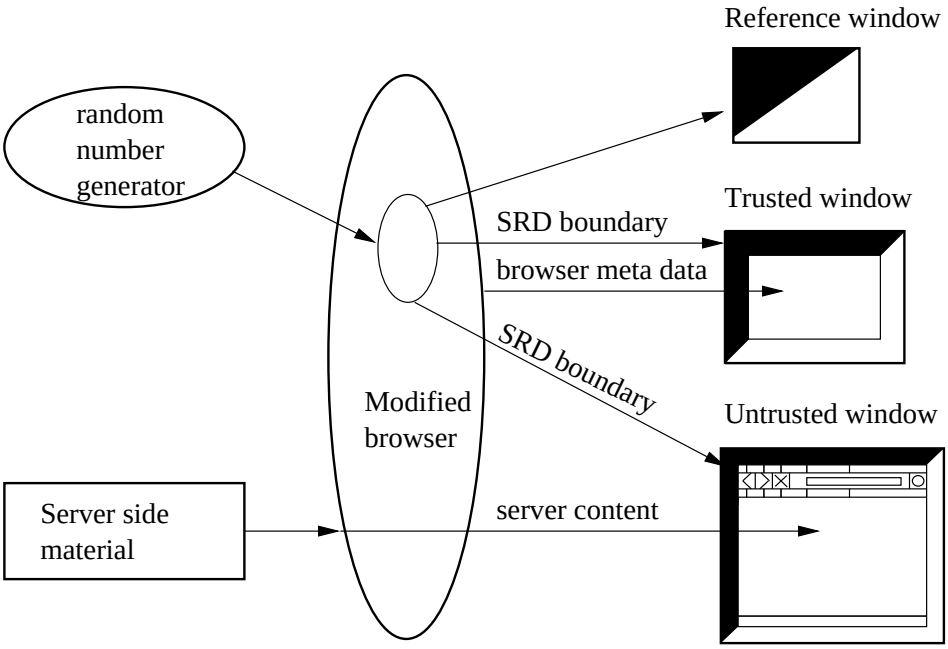


Figure 4.2: The architecture of the SRD approach.

Chapter 5

An Open Source Implementation for SRD Boundary

In order to implement our trusted path solution, we need a browser as its base.

5.1 Choosing Browser 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.

- 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.

- Mozilla is closely related to Netscape, which has a big market share on current desktops.
- Konqueror only run on Linux; Mozilla is able to execute on several platforms.

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

5.2 Install and Debug Mozilla

Mozilla.org makes a tarball of source code every night as the overnight version. They also release stable version of source code at given times. The overnight version of Mozilla and recently released version both can be downloaded at `www.mozilla.org`. Considering the overnight version not to be stable enough, we used Mozilla-0.9.2.1, which was the most recently released version when we started the experiment.

The machine is Dell Optiplex GX 200. The hardware configuration is Intel Pentium III, 800MHz, 256M memory. The operating system is Redhat Linux 6.2.

The installation of Mozilla is pretty straightforward, if all the required tools are installed in advance. If there are some errors, it is probably because the *tar.gz* file or *bgz* file has been corrupted during download.

The Mozilla source code is huge. The tarball is 36.7 MB, more than 20,000 files. The initial time compilation takes one and half hours. Subsequent compilations compilation may takes 5 to 10 minutes, depending on how many files been changed.

Mozilla.org has detailed instructions on how to compile Mozilla. There are Makefiles under the first level subdirectories. If just changing files in one subdirectory, recompiling that subdirectory is enough.

Mozilla is a multi-threaded program; its debugging is very tricky. Compile Mozilla using the command:

```
gmake -f client.mk build
```

to build a debug version Mozilla. The debugger needs so much memory that less than 256MB memory is not recommended.

Mozilla.org has information on how to debug Mozilla. We collect some information here, in order to provide the following work a handy guideline.

- **Launch gdb**: Enter *./dist/bin*; run *./run-mozilla.sh /bin/bash*. *Run-mozilla.sh* is a script to load the Mozilla running environment. Running it alone starts Mozilla. In this case, it starts a bash shell. Start Emacs. In Emacs, press *crtl-x,* then number 2. This would divide Emacs into two frames. Click on the frame in which you want to run gdb to get it focused. Press *alt-x*, then type *gdb*, press return. Emacs asks you "*Run gdb (like this):*", type "*gdb mozilla-bin*". There is *(gdb)* prompt at the Emacs frame, indicating gdb has been launched.

- **Debugging**: all gdb commands should be functional, except *crtl-c*, which should return the control to gdb. *Crtl-c* is very slow responded. When pressed twice, *crtl-c* causes gdb to quit. The way get around this is carefully set break points before let Mozilla take the control. You can not set break points in the module which has not been loaded into memory. However, you can set break points when that module starts being loaded into memory. For example, if you want to set break point when the security module starts to load,

  ```
  set env XPCOM_BREAK_ON_LOAD libpippki
  ```

  should do the work. Using *info thread* or *i th* commands shows how many threads running. Use *thread 2* to switch to thread 2.

- **Reading source code**: The Mozilla source tree is so gigantic that it is difficult to find the right place to read. However, *mozilla.org* has a search engine, *lxr.mozilla.org*, that will help you find a file or a phrase in source tree. You also can use UNIX command *find*, which is more accurate but *lxr.mozilla.org* is much faster. The starting point is Mozilla's *main* function in *./xpfe/bootstrap/nsAppRunner.cpp*.

Because the UNIX command *find* is so useful in understanding Mozilla source code, we give an example here. The following command can find phrase "bookmarks" in all cpp files:

```
find . -type f -name ‘‘*.cpp’’ -exec grep ‘‘bookmarks’’ {} \; -print
```

- -*type f* means only search file, no directories;
- -*name* means match the file name with this condition;
- -*exec* enable to start another UNIX command, usually is *grep*. Put {} after the pattern we want to grep and using \; to finish *grep*;
- -*print* means printing out the search result.

5.3 Mozilla Structure

Before we touch the implementation details, let us go through some basic concepts of Mozilla structure which we use in our implementation.

5.3.1 XUL

XUL is an XML-based user-interface language that was especially created for the Mozilla and Netscape 6 application and is used to define its configurable and downloadable user interface, *chrome*. There are some XUL tutorials in `www.xulplanet.com` and `www.mozilla.org`. However, in order to understand XUL completely, we have to understand XML, where the root is.

Here is a small XUL example, a window with a toolbar which contain two buttons.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://navigator/skin/" type="text/css"?>

<window id="my-window" xmlns:html="http://www.w3.org/1999/xhtml"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
   <toolbar id="my-toolbar">
       <hbox id="bar-buttons">
           <button id="back-button" label="Back"/>
           <button id="forward-button" label="Forward"/>
       </hbox>
   </toolbar>
</window>
```

It is required to declare XML version at the beginning of each XUL file.

```
 <?xml version="1.0"?>
```

XML separates markup language from the style definition. The style information which describes how a

XUL document should look when prepared for viewing by a human is included in a separate document called *stylesheet*. In the second line of above example, the stylesheet documentation is linked into by

```
<?xml-stylesheet href="chrome://navigator/skin/" type="text/css"?>
```

The string after *href* is the location of the stylesheet file. *chrome* means Mozilla user interface; *navigator* is the component name; *skin* is the jar package name. Mozilla groups files and pack them into jar files for fast access and for systematic reason.

The third line in the example tells us the root element of this XUL file is a window object and defines several namespaces. Namespace is used to prevent different domains in Internet has different definitions for the same name. The namepaces used here indicate that all names in this file follow the definition made by Mozilla.org. The declaration of namespace starts with *xmlns* follow by a colon.

The line

```
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
```

is the link to the definition of XUL, which is in the file *there.is.only.xul*. This URL is never actually downloaded. Mozilla will recognize this URL internally.

Each tag, like <*window*> or <*button*> is presented as a *Document Object Model (DOM)* object inside the Mozilla layout engine, *Gecko*.

5.3.2 Themes

Mozilla uses *Cascading Style Sheets (CSS)* and images to define how the browser should look. For example, the style sheet may say: “Back” button should be square and red, or it may say “Back” button should use

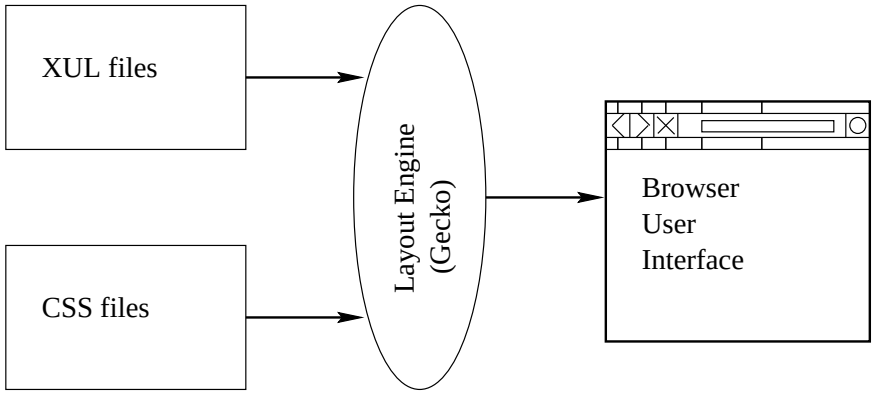


Figure 5.1: The layout engine uses XUL and CSS files to generate the browser user interface.

image “Back.gif”. The collection of the style sheet files is called *skin*. Skins are made popular by MP3 music players, such as Winamp, Sonique, SoundJam, and Macast /citeskin. Skins provide the appearance for these applications, allowing users to customize the appearance of their MP3 player by downloading new skins, or simply creating their own. In Mozilla or Netscape 6, skins are also called *themes*.

Although the appearance changes from skin to skin, the functions of each item do not change, which is defined in XUL files.

Mozilla comes with two themes: modern and classic. They both are in the *themes* directory in source tree.

Figure 5.1 shows the layout engine uses XUL files and CSS files to generate browser user interface.

For more Mozilla themes, check the Web site *http://www.themes.org/skins/mozilla/*.

5.3.3 XPCOM

Mozilla has a modularized structure. Each module is an *XPCOM (Cross Platform Component Object Module)* object and implements a pure virtual C++ interface. XPCOM is a framework for writing cross-platform, modular software. As an application, XPCOM uses a set of core XPCOM libraries to selectively load and manipulate XPCOM components. XPCOM components can be written in C, C++, and JavaScript.

A JavaScript module can directly communicate with 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.

The XPCOM interface is written in *XPIDL (Cross Platform Interface Description Language)* at first. Then the XPIDL file is run through an XPIDL compiler, generate a C++ header file and a typelib file. The C++ header file can be implemented and inherited by other classes.

For more information, `http://www.mozilla.org/projects/xpcom`.

5.3.4 GTK+ toolkit

GTK+ is a free multi-platform toolkit for creating graphical user interfaces, primarily designed for the X Window System. The Linux version of Mozilla uses GTK to create windows and other XUL items, which call the underneath X window functions. GTK+ toolkit is composed of many widgets, such as button, window, bar. Each widget has standard API, like click, drag, press. Linking the API with functions, the widget can response to different mouse actions.

5.4 Implementation details

Implementation of SRD boundaries took three steps. First, we needed to add thicker colored boundaries to all windows. Second, the boundaries needed to change dynamically. Third, the changes needed to happen in a synchronized fashion.

5.4.1 Adding Colored Boundaries

The presence and arrangement of different elements in a window is not hardwired into the application, but is loaded from a separate user interface description, the XUL files. How a XUL element looks is defined by the stylesheets.

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 <*window*> tag in navigator.xul file to be <*externalwindow*>. *Navigator.xul* describe the main surfing window.

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 window as well as the windows opened by server content would be *external windows* with blue boundaries.

5.4.2 Making the Boundaries Dynamic

An XUL element is presented as a DOM object inside the layout engine. The DOM object can have *attributes*. When the attribute is set by JavaScript, the element can be displayed with different style. We gave *window* elements an extra attribute, *borderStyle*. When *borderstyle* attribute is set, *window* element is displayed in different style. This different style also is defined in global.css file.

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

When borderStyle is set as true, the boundary style is *outset*; when borderStyle is “removed”, the boundary style is *inset* . Mozilla notices changes in attributes and updates the repaint the border 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

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

method to get the reference of window. Then use

```
windowRef.setAttribute("borderStyle",true );
```

to set *borderstyle* attribute, and

```
windowRef.removeAttribute("borderStyle");
```

to remove the attribute.

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 only markup the arrangement of items, the responses to the events like click or drag are defined in JavaScript files and link into XUL files. We placed the attribute-changing JavaScript into a separate *js* file and linked it into each corresponding XUL file.

As an 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

```
  windowRef = document.getElementById("example-window");
  windowRef.setAttribute("borderstyle", true);
```

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 JavaScript 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 in XUL file 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.

5.4.3 Adding Synchronization

All the JavaScript files need to look at the same random number, in order to make all windows changing 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.

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 "read-only" string. Read-only means the string only can be read by JavaScript, but can not be set by JavaScript.

```
# include "nsISupports.idl"
[scriptable, uuid(70ab680d-a6a9-4b6e-a3c5-18504783925a)]
interface nsIBorderStyle : nsISupports
{ readonly attribute string value;
};
```

nsISupports.idl is the base interface of all XPCOM module.

The XPIDL compiler transforms this IDL file 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, except the public function, *GetValue*, 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.

Pseudocode for borderstyle module:

```
#define Time_Interval = 5 sec;
```

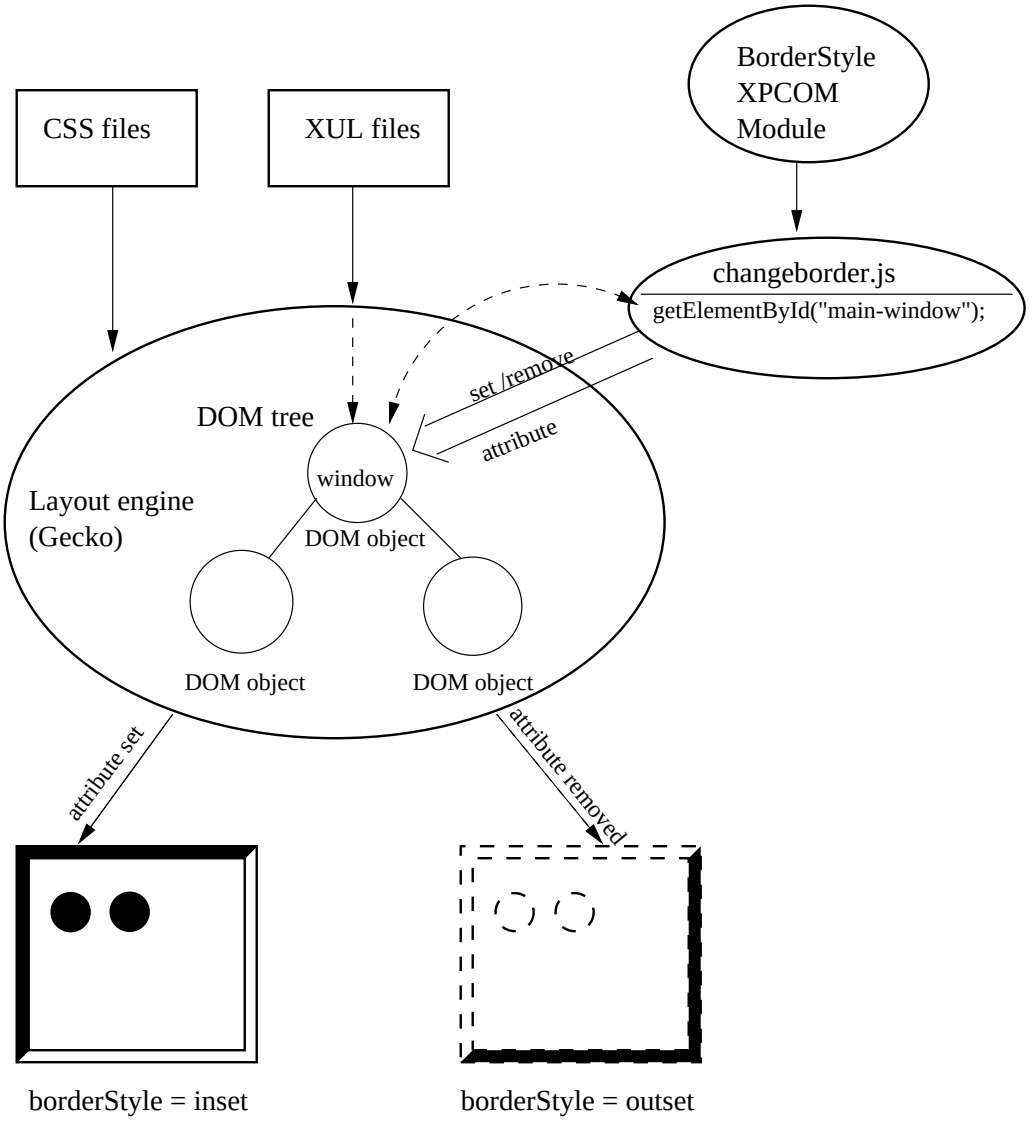


Figure 5.2: This diagram shows the overall structure of our implementation of SRD in Mozilla. The Mozilla layout engine takes XUL files as input, and construct a DOM tree. The root of the tree is the window object. For each window object, JavaScript reads the random number from borderStyle module, and 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.

```
random_number = 0;
timeOne = current_time();

public function GetValue()
{
if ( current_time() >= timeOne + Time_Interval )
setValue();

return random_number;
}

private function setValue()
{
random_number = generateValue();
timeone=time(NULL)+Time_Interval;
}

private function generateValue()
{
temp = read( ‘‘/dev/random’’ );
/* we may use other random number generator here */
return temp;
}
```

GenerateValue function uses Linux kernel random number generator */dev/random*. According to standard Linux documentation [15], this random number generator gathers environmental noise from device drivers and other sources into an entropy pool. When accessed as */dev/random*, random bytes are only returned within the estimated number of bits of noise in the entropy pool (when the entropy pool is empty, the call blocks until additional environmental noise is gathered). When accessed as */dev/urandom*, as many bytes as are requested are returned even when the entropy pool is exhausted.

Every time a thread checks the random value true or false, BorderStyle module compare current time with *timeone*. If current time is bigger than *timeone*, *setValue* is called and ask for a new random number from *generatValue*. If the new generated random number is different from the old one, the *random value* is reset.

At the end of *setValue*, we set

```
timeone= time(NULL)+ Time_Interval;
```

Every *Time_Interval* seconds, a new random number is generated.

Mozilla dynamically load or unload XPCOM modules during running time. When a module is needed, Mozilla search a registration category to find a pointer to this module, then create an instance of it. The registration and unregistration process are mostly taken care of by *Genericfactory* class. But every XPCOM module need a factory class to glue the *Genericfactory* class with itself. *nsBorderStyleModule.cpp* is the factory class of *nsBorderStyleImp*. All the factory class has similar structure. The source code is in Appendix C.

The new borderStyle directory need to be added into Makefiles of *xpcom* directory, in order to be compiled.

5.5 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.

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, *ele-*

ment.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 [7]. 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.

5.6 Reference Window

Problem: As discussed in Section 4.6.2, the enter-SSL warning window use *nsPrompt* service, which is a singleton service in Mozilla. When an enter-SSL warning window pops up, all other windows stop responding and become inactive. This 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. So we want to keep at least one window active as a reference for the synchronization.

Solution: We let borderStyle module fork a new process, which use GTK+ toolkit create a reference window. When a new random number is generated, the borderStyle module would pass 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, 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.

Appendix D lists the source code of referenceWindow.cpp.

5.7 Shell Script

Each window is described by a XUL file. There are hundreds of them. Changing those files one by one and link them with JavaScript file is time consuming, so we prepared shell scripts to do this work automatically. Shell scripts also would add the new generated files into corresponding jar.mn files, in order to pack the js files into the jar package.

Because Mozilla source code changes everyday, our shell scripts only work on certain version of Mozilla, Mozilla-0.9.2.1.

5.8 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 [1] 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.

5.9 Discussion

5.9.1 Inner and outer SRD

Some browser windows, like the main surfing 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.

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* respectively to denote these two approaches.

Inner SRD raises some complicated questions. As an example, that every element in a window marks up by colored boundary weakens satisfaction of the minimal intrusiveness design criteria. An alternative model would be assuming every element inside a trusted window is trusted, so does not need inner SRD boundary. Only the trusted element inside an untrusted window would need a SRD boundary. However, this approach will complicate the security semantics and may confuse the user to make trust decision.

5.9.2 Comparison with Compartmented Model Workstations

In our defense against Web-spoofing, we mark windows with different security levels according to the material they hold, and try to build a user interface that clearly indicate these levels. The idea is similar with Compartmented Model Workstations (CMW).

However, we are different from CMW in how to indicate the genuineness of windows. CMW uses a reserved color label to indicate the secure level of the window which gets focus currently. If it is a spoofed window, the reference color label would not be consistent with the window color label. CMW approach separate the *status* information from the *content* information in space and time. Therefore, the *status* information is easy to be neglected by the careless users. In SRD boundaries approach,the user checks whether the window's boundary changing synchronously with reference window or other windows. The *status* information and *content* information are bound together. The users receive both of them simultaneously.

CMW approach need the user click on the window to get it focus to check its genuineness. SRD boundaries approach bring the *status* information to the users automatically. The only thing the users need do is observation.

The advantage of CMW approach is less distracting, because most of the labels are static, only the reference color label is changing. However, less distracting also means winning less attention. SRD boundary attracts more user's attention. Therefore, it is easy for the user notice if something goes wrong.

The key point SRD boundaries introduced into the UI design of Web browser is the random dynamic parameter. This random dynamic parameter does not have to be constructed by the boundaries with different styles. It can be a bar with progressively changing or some other dynamic designs. A careful design may reduce the irritating effect brought by the dramatic changing of boundaries, therefore, bringing the key idea of SRD boundaries into practical use.

5.9.3 Some Thoughts about Trusted and Untrusted Material

In SRD boundaries design, we have a clean and simple definition of trusted and untrusted material. The material from the server is untrusted, so it presented in blue boundary windows; the information from the browser is trusted, so it is presented in orange boundary windows. However, a deeper exploration of trusted and untrusted material raises some interesting thoughts.

We should trust some server delivered material at least, for example, the page source. It is from the server, but is presented plainly by the browser, the attacker can not play tricks on it. We should trust the page source information. However, according to our definition, it should not be trusted.

If we define the trusted material as the material from the browser and the plain text from server, the server pops up an alert window only contains a single string, should we trust it? Is it possible that a server alert window can be used to spoof an entering-SSL warning windows? At least, an alert window which is popped

up by the server but contains the same text as an entering-SSL window would confuse the user.

So perhaps we should define the untrusted material as material from the server which presented by the browser but may confuse the user. This definition actually is as ambiguous as untrusted material itself.

If we define that all windows popped by the server are untrusted, and any windows popped up the browser are trusted, then is the prompt window popped up by server during client authentication trusted or not? If it is untrusted, why should the user type a password into it? If it is trusted, it is popped up by server.

Another way to define trusted material is only status information from browser is trusted, anything else is untrusted. In this thesis, we defined the status information as the information that expressed by the browser user interface and indicates the connection status between the browser and the server. Does the server certificate belong to the status information? Thinking over this question may bring up more interesting thoughts on what is trusted or untrusted material.

Chapter 6

SRD Boundaries 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 obtain 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.

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.

Chapter 7

Summary

The server-client communication is ended in users instead of browsers. However, it is the browser instead of the user that receives all the server information. The user needs to make trust judgments based on the user interface signals presented by the browser. This is why there must be clear and correct understanding of these browser signals by the user. However, our survey on modern browsers and our Web spoofing work show that the information collected and presented by browsers are neither sufficient nor reliable.

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; it 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 Synchronized random dynamic (SRD) solution obtains this unpredictability from randomness.

SRD boundaries are different color boundaries around windows, marking the security level of windows. The server content material is contained by a blue boundary window, the browser material is contained by a brown boundary window. All these boundaries dynamically change their styles according to a random number generator result, either to be inset or be outset.

We implemented SRD boundaries based on Mozilla source code. An XPCOM module uses Linux kernel random number generator and generates random number. JavaScript reads the random number and changes the window attribute accordingly. A different attribute brings different border style to the window.

The fact that pop up windows, such as entering-SSL window, deactivate all other windows, make synchronization inconsistent at this point. We designed a reference window which will always be active in synchronization even when the pop up window starts. What we did is to let the borderstyle XPCOM module invoke another reference window process which displays images indicating inset or outset boundary style . The random number generator result is written into the pipe to the reference window process which uses GTK+ graphic toolkit.

The user study results show that SRD approach can be learned easily.

Because of the design pattern of Mozilla source code, there are a few known bugs which we are working on fixing.

Chapter 8

Future work

Besides the known deficiencies discussed in Section 5.8, we also are considering other areas for future work.

Install SRD boundary in other modules So far, we only installed SRD in

- the main navigator window
- the preference window
- the Page info window
- the Page source window.

Some other modules should be considered to install SRD as well. The primary candidate is *Personal Security Module (PSM)*—the module taking care of certificates and password which specially need to be protected.

Buffer the random bits We access the random number generator as /dev/random, which means if the entropy pool is empty, the call for new random number would not return. As we mentioned before, if an

XPCOM module is a singleton service. When one thread is accessing it, the other threads are blocked. If the call does not return promptly, this may potentially slow down overall performance. One way to solve this problem is read out certain bits one time and buffer them for later use.

Other platforms Mozilla can run on different platforms. We chose Linux, because Linux is a major platform in the research world. Windows is more dominant in society at large, so we need to port our modifications there. This will require modifications to our Linux-specific techniques, such as /dev/random for randomness and GTK for the reference window.

Multiple SSL As Chapter 2, we discussed that no current Web browser can display the fact that multiple certificates are in use simultaneously. If a page in Server A contains SSL session to Server B, due to the browser's failure to indicate the existence of Server B SSL session, there is a potential for deny service attack to Server B. Although multiple SSL sessions rarely happen, it is important for browsers to be able to display multiple certificates for completeness sake.

Also as mentioned in introduction, how does browse tell the user when there is a delegation relation? For example, Modus Media legitimately represents Palm Computing. How does the server certificate structure include some way to indicate it? This will be a interesting new research topic.

Server information collection So far, server can collect all kinds of information from browsers, such as the browser name, version, support Javascript or not. However, there is little information the browser knows about the server. If the user knows the server information, like what kind of security policy the server employs and on what kind of hardware, he can decide whether the server meets his security requirement and whether he wants to continue the communication. This will be another interesting area to be work on.

Visual Hashes. In personal communication, Perrig suggests using visual hash information [12] 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.

Appendix A

Source code of *nsBorderStyle.h*

```
# include "nsIBorderStyle.h"
# include <time.h>

/*  using uuidgen generated a unique identify which hopefully doesn’t
 * conflict with other uuid.
 * CID for this class is
 * {478803ab-c024-43e4-a8ff-7a339d486185}
 */

# define NS_BORDERSTYLE_CID \
{ 0x478803, 0xc024, 0x43e4, { 0xa8, 0xff, 0x7a, 0x33, 0x9d, 0x48, 0x61, 0x85} }

# define NS_BORDERSTYLE_CONTRACTID "@mozilla.org/borderstyle;1"

class nsBorderStyleImpl : public nsIBorderStyle
{
 public:
  nsBorderStyleImpl();
  virtual ~nsBorderStyleImpl();

  /* This macro expands into a declaration of the nsISupports interface.
   * Every XPCOM component needs to implement nsISupports, as it acts
   * as the gateway to other interfaces this component implements
   * QueryInterface, AddRef, and Release.*/
  NS_DECL_ISUPPORTS

  /* This macro defined in nsIBorderStyle.h
   * NS_IMETHOD GetValue(char * *aValue); */
  NS_DECL_NSIBORDERSTYLE

  private:
```

```
  /* function generateValue is used for generating random numer 0 or 1 */
  void generateValue();

  /* function to set Value according the randome generator */
  void setValue();

  char * mValue;
  int randomNumber1;  // a number to remember the old random number
  int randomNumber2;  // a number to hold the randome generator result
  time_t t1;          // used to record current time.
};
```

Appendix B

Source code of *nsBorderStyle.cpp*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/state>
#include <fcntl.h>
#include <unistd.h>
#include "plstr.h"
#include "nsBorderStyle.h"
#include "nsMemory.h"

nsBorderStyleImpl::nsBorderStyleImpl()
{
  NS_INIT_REFCNT();
  mValue= PL_strdup("false");
  randomNumber1 =0;
  randomNumber2 =0;
  t1=time(NULL);

  /* create a pipe */
  if (pipe(fd)<0)
    printf("pipe error\n");

  /* start a new process to run  gtk program */
  if ( (pid = fork())<0)
    printf("fork error\n");

  else if (pid >0){   /* parent process generate random number */
    close(fd[0]);
    write(fd[1],"false\n", 6);
  }
  else{               /* child process */
```

```
    close(fd[1]);     /*close child process' write end. */

      if (fd[0] != STDIN_FILENO){
        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
     printf(" dup2 error\n");
        close( fd[0]);    }

   /* start to run the gtk program */
   if ( execl("./referenceWindow/reference_window",
              "./referenceWindow/reference_window", NULL) <0)
     printf(" execl error\n");

    /* exit child. note the use of _exit() instead of exit() */
     _exit(-1);
   }
}

nsBorderStyleImpl::~nsBorderStyleImpl()
{
  if (mValue) PL_strfree(mValue);
}

/* NS_IMPL_ISUPPORTS1_CI is a macro which is defined in
 * nsISupportsImpl.h#740, not in #700, the lxr result is wrong!
 *
 *  #define NS_IMPL_ISUPPORTS1_CI(_class, _interface)
 *   NS_IMPL_ADDREF(_class)
 *   NS_IMPL_RELEASE(_class)
 *   NS_IMPL_QUERY_INTERFACE1_CI(_class, _interface)
 *   NS_IMPL_CI_INTERFACE_GETTER1(_class, _interface)*/
NS_IMPL_ISUPPORTS1_CI(nsBorderStyleImpl, nsIBorderStyle);

/* Notice that in the protoype for this function, the NS_IMETHOD macro was
 * used to declare the return type.  For the implementation, the return
 * type is declared by NS_IMETHODIMP
 * this function's implementation is as same as in nsSample.cpp */
NS_IMETHODIMP nsBorderStyleImpl::GetValue(char * *aValue)
{
  /* if the timeNow is bigger than t1, that means a new random
   * number need be generated.*/
  if (time(NULL)>= t1)
    setValue();

    NS_PRECONDITION(aValue != nsnull, "null ptr");
    if (! aValue)
        return NS_ERROR_NULL_POINTER;

    if (mValue) {
         *aValue = (char*) nsMemory::Alloc(PL_strlen(mValue) + 1);
        if (! *aValue)
            return NS_ERROR_NULL_POINTER;
    PL_strcpy(*aValue, mValue); }
    else { *aValue = nsnull;}
    return NS_OK;
```

```
}

void nsBorderStyleImpl::generateValue()
{
  unsigned char readOut; // readOut hold the unsigned char
 //value which read out from "/dev/random"
  int count;
  int fd;

  fd = open("/dev/random", O_RDONLY);

  if (-1 == fd) {
    fprintf(stderr,"error in open dev!\n");
    exit(-1);  }

  count = read(fd, &readOut, 1);

  if (count != 1) {
    fprintf(stderr,"error in read dev!\n");
    exit(-1);  }

  randomNumber2= (int)readOut%8/4;  //convert readOut to 1 or 0
  close(fd);
}

void nsBorderStyleImpl::setValue()
{
  generateValue();
        if( randomNumber1 != randomNumber2){
      /* The new generate random number is different
       * record the new randomNumber in randomNumber 1 for next
       * cycle use.*/
      randomNumber1= randomNumber2;

      if (mValue){ PL_strfree(mValue);}

      if ( randomNumber1 ==0){ mValue = PL_strdup("true");}
      else{ mValue = PL_strdup("false");}
        }
t1=time(NULL)+2;
}
```

Appendix C

Source code of *nsBorderStylModule.cpp*

```
#include "nsIGenericFactory.h"
#include "nsBorderStyle.h"

////////////////////////////////////////////////////////////////////////
// With the below sample, you can define an implementation glue
// that talks with xpcom for creation of component nsBorderStyleImpl
// that implement the interface nsIBorderStyle. This can be extended for
// any number of components.
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Define the contructor function for the object nsBorderStyleImpl
//
// What this does is defines a function nsBorderStyleImplConstructor which we
// will specific in the nsModuleComponentInfo table. This function will
// be used by the generic factory to create an instance of nsBorderStyleImpl.
//
// NOTE: This creates an instance of nsBorderStyleImpl by using the default
//  constructor nsBorderStyleImpl::nsBorderStyleImpl()
//
NS_GENERIC_FACTORY_CONSTRUCTOR(nsBorderStyleImpl)

////////////////////////////////////////////////////////////////////////
// Define a table of CIDs implemented by this module along with other
// information like the function to create an instance, contractid, and
// class name.
//
// The Registration and Unregistration proc are optional in the structure.
//
static NS_METHOD nsBorderStyleRegistrationProc(nsIComponentManager *aCompMgr,
                                               nsIFile *aPath,
```

```
                                        const char *registryLocation,
                                        const char *componentType,
                                        const nsModuleComponentInfo *info)
{
    // Do any registration specific activity like adding yourself to a
    // category. The Generic Module will take care of registering your
    // component with xpcom. You dont need to do that. Only any component
    // specific additional activity needs to be done here.

    // This functions is optional.

    return NS_OK;
}

static NS_METHOD nsBorderStyleUnregistrationProc(nsIComponentManager *aCompMgr,
                                          nsIFile *aPath,
                                          const char *registryLocation,
                                          const nsModuleComponentInfo *info)
{
    // Undo any component specific registration like adding yourself to a
    // category here. The Generic Module will take care of unregistering your
    // component from xpcom. You dont need to do that. Only any component
    // specific additional activity needs to be done here.
    // This functions is optional. If you dont need it, dont add it to the structure.

    // Return value is not used from this function.
    return NS_OK;
}

// For each class that wishes to support nsIClassInfo, add a line like this
NS_DECL_CLASSINFO(nsBorderStyleImpl)

static nsModuleComponentInfo components[] =
{
  { "Sample Component", NS_BORDERSTYLE_CID, NS_BORDERSTYLE_CONTRACTID,
    nsBorderStyleImplConstructor,
    nsBorderStyleRegistrationProc /* NULL if you dont need one */,
    nsBorderStyleUnregistrationProc /* NULL if you dont need one */,
    NULL /* no factory destructor */,
    NS_CI_INTERFACE_GETTER_NAME(nsBorderStyleImpl),
    NULL /* no language helper */,
    &NS_CLASSINFO_NAME(nsBorderStyleImpl)
  }
};

////////////////////////////////////////////////////////////////////////
// Implement the NSGetModule() exported function for your module
// and the entire implementation of the module object.
//
// NOTE: If you want to use the module shutdown to release any
// module specific resources, use the macro
// NS_IMPL_NSGETMODULE_WITH_DTOR() instead of the vanilla
// NS_IMPL_NSGETMODULE()
//
NS_IMPL_NSGETMODULE(nsBorderStyleModule, components)
```

Appendix D

Source code of *referenceWindow.cpp*

```
 /* reference window is the window pop up by borderstyle module.
 * so when enter_ssl warning window blocks all other threads,
 * the reference window still be active to provide the right border
 * style reference for the user.
 *
 * a gtk program compiler with
 * gcc -o reference_window reference_window.c ‘gtk-config --cflags --libs‘
 *
 * in order to let execl work, you have to start mozilla process
 * by entering ./dist/bin/
 * then start mozilla by ./mozilla
 */

#include <gtk/gtk.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* viewport has two adjustments */
GtkObject *adj1, *adj2;

/*  call back function for window close
 *  window close will terminate the process */
gint close_application( GtkWidget *widget,
                        GdkEvent  *event,
                        gpointer   data )
{   gtk_main_quit();
    return(FALSE);
}
```

```
/* callback function for monitoring pipe.
 * Pipe's writing end is borderStyle module.
 * This is pipe's reading end
 * read out from pipe and adjust the window viewport */
static void input_callback( gpointer data, gint fd,
    GdkInputCondition condition)
{ int n;
  char line[10];
  char *true="true\0";
  char *false="false\0";

  /* read out from pipe, we used dup2 map fd[0] with STDIN_FILENO
   * so here, we read from STDIN_FILENO
   */
  n = read(STDIN_FILENO, line, 20);

  if (strncmp( (char*)&line, true, 5) >0){
     gtk_adjustment_set_value( (GtkAdjustment *)adj2, 245.0 ); }
  else{
      gtk_adjustment_set_value( (GtkAdjustment *)adj2, 0.0 );  }
}

/* the main function for create the window */
void creat_window(void)
{ int i, j;
  GtkWidget *window;
  GtkWidget *view;
  GtkWidget *table;
  GtkWidget *picture_left;
  GtkWidget *picture_right;
  GdkPixmap *right_image_pixmap;
  GdkPixmap *left_image_pixmap;

  /* GdkBitmap for structures of type GdkPixmap,*/
  GdkBitmap *mask_left;
  GdkBitmap *mask_right;
  GtkStyle *style;

  /* create a new window*/
  window = gtk_window_new (GTK_WINDOW_DIALOG);

  /* limit window size*/
  gtk_widget_set_usize(window, 145, 130);

  /* after link to this singal, window close will treminate the process*/
  gtk_signal_connect( GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (close_application), NULL );

  /* window border width set*/
  gtk_container_set_border_width( GTK_CONTAINER (window), 2 );

  /* Create a 1x3 table */
  table = gtk_table_new (3, 1, TRUE);

  /* now for the pixmap from gdk */
```

```
  style = gtk_widget_get_style( window );

  /* get the GdkPixmap from the file.*/
  left_image_pixmap = gdk_pixmap_colormap_create_from_xpm( window->window,
       NULL, &mask_left, &style->bg[GTK_STATE_NORMAL], "./left.xpm");
  right_image_pixmap = gdk_pixmap_colormap_create_from_xpm( window->window,
       NULL, &mask_right, &style->bg[GTK_STATE_NORMAL], "./right.xpm");

  /* get the Gtk image widget from the pixmap, */
  picture_left = gtk_pixmap_new ( left_image_pixmap, mask_left);
  picture_right = gtk_pixmap_new ( right_image_pixmap, mask_right);

  /* Insert picture_left into the upper left quadrant of the table */
  gtk_table_attach_defaults (GTK_TABLE(table), picture_left, 0, 1, 0, 1);

  /* Insert picture_right into the down left quadrant of the table */
  gtk_table_attach_defaults (GTK_TABLE(table), picture_right, 0, 1, 2, 3);

  /* create two adjustments for viewport */
  adj1 = gtk_adjustment_new (0.0, 0.0, 101.0, 1, 10.0, 10.0);
  adj2 = gtk_adjustment_new (0.0, 0.0, 101.0, 1, 10.0, 10.0);

  /* create the viewport */
  view = gtk_viewport_new( (GtkAdjustment *)adj1, (GtkAdjustment *)adj2 );

  /* let viewport contain the table */
  gtk_container_add (GTK_CONTAINER (view), table);

  /* let the window contain the viewport */
  gtk_container_add (GTK_CONTAINER (window), view);

  gtk_widget_show  (window);
  gtk_widget_show  (view);
  gtk_widget_show  (table);
  gtk_widget_show  (picture_left);
  gtk_widget_show  (picture_right);

  /* monitoring IO, when the pipe is ready for reading,
   * input_callback is called
   */

  gdk_input_add( STDIN_FILENO, GDK_INPUT_READ,
 input_callback, NULL  );
}

int main( int argc,
  char * argv[])
{
  gtk_init(&argc, &argv); //setup gtk environment
  creat_window();
  gtk_main();
  return(0);
}
```

Bibliography

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

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

[3] D. Dean and D. Wallach. Personal communication, Summer 2001.

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

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

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

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

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

[9] Alex Lee “Skins, themes and chromes explained” `http://www.gerbilbox.com/newzilla/netscape6/ui01.php` Updated on Aug 16, 2000

[10] Opera company Web site `http://www.opera.com`

[11] Opera’s history page `http://www.opera.com/company/presentation/history.html`

[12] 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)*. pp. 131-138. July 1999.

[13] F. De Paoli, A. L. DosSantos and R. A. Kemmerer. “Vulnerability of ‘Secure’ Web Browsers.” *Proceedings of the National Information Systems Security Conference.* pp. 476-487, October 1997.

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

[15] Secure Programming for Linux and Unix HOWTO `http://www.tldp.org/HOWTO/Secure-Programs-HOWTO/random-numbers.html`

[16] *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`

[17] Bob 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`)

[18] Sun Microsystem Document: Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated? `http://java.sun.com/j2se/1.3/docs/guide/misc/threadPrimitiveDeprecation.html`

[19] The Mozilla organization `http://www.mozilla.org`

[20] The K desktop environment organization `http://www.kde.org`

[21] Dan Tobias “Dan’s Web tips” `http://webtips.dantobias.com/brand-x/index.html`

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

[23] The Browser Spy Web site `http://gemal.dk/browserspy`

[24] “Uniform Resource Locators (URL)” `www.rfc.net/rfc1738.html` December, 1994

[25] E.Z.Ye, Y, Yuan, S.W.Smith,“Web Spoofing Revisited: SSL and beyond”, Technical Report TR2002-417, Department of Computer Science, Dartmouth College

[26] E. Ye, S.W. Smith, “Trusted Paths for Browsers: An Open-Source Solution to Web Spoofing” Technical Report TR2002-418, Department of Computer Science, Dartmouth College

[27] E.Z. Ye, S.W. Smith “Trusted Paths of Web Browsers” accepted by 11th USENIX Security Symposium (Security ’02)