This is a version after the camera-ready version in the IEEE S&P'09 proceedings, but before the conference presentation. In this version, we add Apple's response regarding Safari, and update the progress by other browser vendors.

Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments

Shuo Chen
Microsoft Research
Redmond, WA, USA
shuochen@microsoft.com

Ziqing Mao
Purdue University
West Lafayette, IN, USA
zmao@cs.purdue.edu

Yi-Min Wang
Microsoft Research
Redmond, WA, USA
ymwang@microsoft.com

Ming Zhang
Microsoft Research
Redmond, WA, USA
mzh@microsoft.com

***Abstract* – HTTPS is designed to provide secure web communications over insecure networks. The protocol itself has been rigorously designed and evaluated by assuming the network as an adversary. This paper is motivated by our curiosity about whether such an adversary has been carefully examined when HTTPS is integrated into the browser/web systems. We focus on a specific adversary named "Pretty-Bad-Proxy" (PBP). PBP is a malicious proxy targeting browsers' rendering modules above the HTTP/HTTPS layer. It attempts to break the end-to-end security guarantees of HTTPS without breaking any cryptographic scheme. We discovered a set of vulnerabilities exploitable by a PBP: in many realistic network environments where attackers can sniff the browser traffic, they can steal sensitive data from an HTTPS server, fake an HTTPS page and impersonate an authenticated user to access an HTTPS server. These vulnerabilities reflect the neglects in the design of modern browsers – they affect all major browsers and a large number of websites. We believe that the PBP adversary has not been rigorously examined in the browser/web industry. The vendors of the affected browsers have all confirmed the vulnerabilities reported in this paper. Most of them have patched or planned on patching their browsers. We believe the attack scenarios described in this paper may only be a subset of the vulnerabilities under PBP. Thus further (and more rigorous) evaluations of the HTTPS deployments in browsers appear to be necessary.**

Keywords: pretty-bad-proxy, HTTPS deployment, browser security

I. INTRODUCTION

HTTPS is an end-to-end cryptographic protocol for securing web traffic over insecure networks. Authenticity and confidentiality are the basic promises of HTTPS. When a client communicates with a web server using HTTPS, we expect that: i) no HTTPS payload data can be obtained by a malicious host on the network; ii) the server indeed bears the identity shown in the certificate; and iii) no malicious host in the network can impersonate an authenticated user to access the server. These properties should hold as long as the end systems, i.e. the browser and the server, are trusted.

In other words, the adversary model of HTTPS is simple and clear: the network is completely owned by the adversary, meaning that no network device on the network is assumed trustworthy. The protocol is rigorously designed, implemented and validated using this adversary model. If HTTPS is not robust against this adversary, it is broken by definition.

This paper is motivated by our curiosity about whether the same adversary that is carefully considered in the design of HTTPS is also rigorously examined when HTTPS is integrated into the browser. In particular, we focus on an adversary called "Pretty-Bad-Proxy" (PBP), which is a man-in-the-middle attacker that specifically targets the browser's rendering modules above the HTTP/HTTPS layer in order to break the end-to-end security of HTTPS. Figure 1 illustrates this adversary: PBP can access the raw traffic of the browser (encrypted and unencrypted), but it is unable to decrypt the encrypted data on the network. Instead, the PBP's strategy is to send malicious contents through the unencrypted channel into the rendering modules, attempting to access/forge sensitive data (which flow in the encrypted channel on the network) above the cryptography of HTTPS.

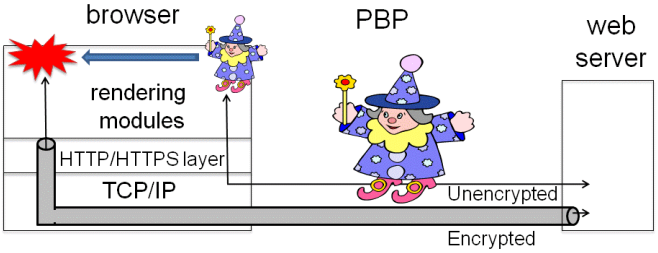


Figure 1. PBP attacks the encrypted data after they are decrypted above the HTTPS layer

With a focused examination of the PBP adversary against various browser behaviors, we realize that PBP is indeed a threat to the effectiveness of HTTPS deployments. We have discovered a set of PBP-exploitable vulnerabilities[1] in IE, Firefox, Opera, Chrome browsers and many websites. They are due to a number of subtle behaviors of the HTML engine, the scripting engine, the HTTP proxying, and the cookie management. By exploiting the vulnerabilities, a PBP can obtain the sensitive data from the HTTPS server. It can also certify malicious web pages and impersonate authenticated users to access the HTTPS server. Although all attacks fool the HTTP/HTTPS layer and above, the manifestations of the vulnerabilities are diversified: some require the scripting capability of the browser while others use static HTML contents entirely;

[1] All vulnerabilities exploitable by the PBP exist on or above the HTTP/HTTPS layer. Technically speaking, some of the vulnerabilities can also be exploited by a router or a switch as well, although the IP layer and the link layer are not the culprits.

some require the HTTP-proxy mechanism enabled in the browser while others do not need this requirement. The existence of the vulnerabilities clearly undermines the end-to-end security guarantees of HTTPS.

People who are less familiar with HTTPS sometimes argue that the HTTPS security inherently depended on the trust on the proxy, and thus the assumption about a malicious proxy was inappropriate. This argument is conceptually incorrect since HTTPS' goal is to achieve the end-to-end security. Also, we show that in practice the trust on the proxy is too brittle for HTTPS to depend on. We constructed two versions of attack programs to show two levels of threats: (1) the first level, which is already serious, is due to the wide use of proxies for web access. The integrity of proxies is generally difficult to ensure. For instance, malware and attackers may take over legitimate proxies in hotels and Internet cafes, because they are not well managed. Many free third-party open proxies are also essentially unaccountable, etc; (2) the second level, which is more severe, is due to the fact that browsers' proxy-configuration mechanisms and browsers' communications with proxies are often unencrypted in many network environments. This makes a user vulnerable even when he/she is not knowingly connected to an untrusted proxy, as long as an attacker has the MAC layer access to the victim's network. In our Ethernet and WiFi experiments, the attacker simply needs to connect to the same Ethernet local area network (LAN) or wireless access point (AP) to launch the attacks. The damages of such attacks are the same as those caused by physically taking over a legitimate proxy. With the PBP vulnerabilities in browsers, the end-to-end security guarantees promised by HTTPS are lost because users basically need to trust the network in order to trust HTTPS.

We have reported the discovered vulnerabilities to browser vendors. They have acknowledged the attack scenarios. The status of vendor responses is given later in the paper in Table III. Most of the vendors have patched or planned on patching their browsers.

A note about this paper: This work was finished in July 2007, except for the paper writing and the vulnerability testing on the Google Chrome browser released in beta in Sept. 2008. The paper submission has been withheld until this conference. To present this work in a necessary context, we will describe how our effort is related to some of the efforts from other researchers in this time frame.

The rest of the paper is organized as follows. Section II introduces the basic concepts about the browser security model and the HTTPS protocol. Section III and Section IV describe various PBP attacks. In section V, we demonstrate the feasibility of exploiting these vulnerabilities and study their security implications in real-world settings. Section VI discusses possible fixes and mitigations. Section VII covers related work and Section VIII concludes.

II. SAME-ORIGIN-POLICY AND HTTPS

A. Same-Origin Policy

Browsers support the functionality of downloading contents and executing scripts from different websites at the same time. Given some websites may contain malicious contents, it is crucial that browsers isolate the contents and scripts of different websites in order to prevent cross-domain interference. In addition, browser should allow scripts to access the contents of the same websites in order to perform normal web functionalities. This access-control policy is referred to as the *same-origin policy*.

Scripts and static contents are rendered and composed into webpages. The same-origin policy is enforced by isolating webpages according to their own security contexts derived from their URLs. A typical URL is represented in the format of "`protocol://serverName:port/path?query`" and the corresponding security context is a three-tuple `<protocol,serverName,port>`. As an example, the `protocol` can be HTTP or HTTPS, the `serverName` can be *www.ebay.com*, and the `port` can be 80, 443, or 8080, etc.

Each webpage is hosted in a frame or an inline frame. A browser window is a top level frame, which hosts the webpage downloaded from the URL shown in the address bar. A webpage can create multiple frames and inline frames to host webpages from different URLs. The access control mechanism between these webpages conforms to the same-origin policy described above. For example, suppose frame `w1` loads a webpage from *https://bank.com* and frame `w2` loads a webpage from *http://bank.com* or *https://evil.com*. If the script running in `w2` attempts to access an HTML object inside `w1`, the access will be denied by the browser's security mechanism because of the same-origin policy. Without the same-origin policy, the document content of *https://bank.com* would be accessible to a script embedded in the webpage from *http://bank.com* (which could be faked by proxies and routers because it is not encrypted) or from *https://evil.com*, which would defeat the purpose of HTTPS.

Similar to frame, other objects, such as `XML` and `XMLHTTPRequest`, rely on the same-origin policy to protect their documents as well. Also, webpages can be attached with a type of plain-text data called cookies. Cookies have a slightly different same-origin policy, which will be described in Section IV.B.

B. Basics of HTTPS and Tunneling

HTTPS is the protocol for HTTP communications over Secure Sockets Layer (SSL) or Transport Layer Security (TLS) [6]. For simplicity, in the rest of the paper, we use "SSL" to refer to both SSL and TLS. HTTPS is widely used to protect sensitive communications, such as online banking and online trading, from eavesdropping and man-in-the-

middle attacks. At the beginning of an HTTPS connection, the browser and the web server go through an SSL handshake phase to ensure that: 1) the browser receives a legitimate certificate of the website issued by a trusted Certificate Authority (CA); and 2) the browser and the server agree on various cryptographic parameters, such as the cipher suite and the master key, in order to secure their connection. Once the handshake succeeds, encrypted data flow between the browser and the server. A malicious proxy or router may disrupt the communication by dropping packets, but it should not be able to eavesdrop or forge data.

All major browsers support HTTPS communications through HTTP proxy. The mechanism is referred to as "tunneling". Before starting the SSL handshake, the browser sends an HTTP CONNECT request to the proxy, indicating the server name and port number. The proxy then maintains two TCP connections, with the browser and with the server, and serves as a forwarder of encrypted data. To tunnel the HTTPS packets between the two TCP connections, the proxy needs to set different values in the IP and TCP headers, such as IP addresses and port numbers. But it is not able to manipulate the encrypted payload besides copying it byte-by-byte. Therefore, the proxy does not have any additional information about HTTPS traffic beyond the IP and TCP headers. Normally an adversary must break the cryptographic schemes used by HTTPS in order to access the actual HTTPS contents. Note that a proxy is not a trusted entity in HTTPS communications. By design, confidentiality and authenticity of HTTPS should be guaranteed when the traffic is tunneled through an untrusted proxy; in reality, as we will show in Section V, proxies are widely used in many network environments where proxies are not expected to be trustworthy. Being merely an interconnecting host on the network, the proxy is not a trusted entity that the HTTPS security relies on.

In the next two sections, we describe PBP attack scenarios. The versions of the browsers in our discussion are IE 7, IE 8, Firefox 2, Firefox 3, Safari 3, Opera 9, Chrome Beta and Chrome 1.

III. SCRIPT-BASED PBP EXPLOITS

Scripting is a critical capability of modern browsers. However, they impose more risks than static HTML contents if the scripting mechanism is not carefully designed and evaluated against different types of adversaries. Cross-site scripting [13] and browser cross-domain attacks [4] are the representative examples of vulnerabilities exposed by scripting. While these attacks have provoked many discussions in the web security community, so far there has been less attention on the possibility of script-based attacks against HTTPS when the proxy is assumed the adversary.

In this section, we will describe several script-based attacks, some of which are because of executing regular HTTP scripts in the HTTPS context while others are because of executing scripts from unintended HTTPS websites in the context of target HTTPS websites. These attacks raise a concern that browsers' scripting mechanisms have not been thoroughly examined under the PBP adversary.

A. Embedding Scripts in Error Responses

We explained earlier that the browser sends an HTTP CONNECT request to the proxy when it tries to access an HTTPS server through the proxy. Sometimes the proxy may fail in connecting to the target server, in which case the proxy should send an HTTP error message back to the browser. For instance, when the browser requests *https://NonExistentServer.com*, the proxy will return an HTTP 502 Proxy Error message to the browser because the proxy cannot find a valid IP address associated with the server name *NonExistentServer.com.* Note that the communication between the browser and the proxy still uses plain-text up to this point. Interestingly enough, the browser renders the error response in the context of *https://NonExistentDomain.com*, although the server does not exist. We observed this behavior on all browsers that we studied. In addition to HTTP 502, other HTTP 4xx and 5xx messages are treated in a similar way.

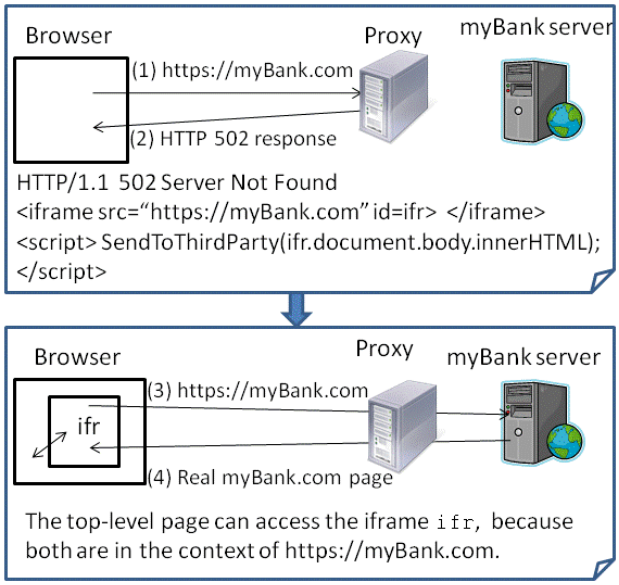


Figure 2. Embedding scripts in 4xx/5xx error messages

Since the browser completely relies on the proxy for the tunneling, the proxy can arbitrarily lie to the browser, which leads to the compromise of HTTPS confidentiality and authenticity. We now use an example to illustrate how a PBP adversary can steal the sensitive data from the browser when it is visiting an HTTPS server. Suppose the browser is accessing *https://myBank.com*, upon receiving the HTTP CONNECT request from the browser, the proxy may pretend that the server did not exist by returning an HTTP 502 error message. The error message also includes an iframe (inline frame) and a script. When the browser

renders the error message, the iframe will cause the browser to send another CONNECT request for *https://myBank.com*. The proxy will behave normally this time by tunneling the communication to the server. Thereafter, user's banking data will be loaded into the iframe (abbreviated as `ifr`). However, the script embedded in the original error message has been running in the context of *https://myBank.com*. This allows the script to reference `ifr.document` and send the user's banking data (e.g., `body.innerHTML`) to a third party machine, circumventing the same-origin policy of the browser. Besides peeking the user's banking data, the attacker can also transfer money from the bank on behalf of the user.

The attack does not depend on which authentication mechanism is used between the victim and the server. For instance, if the server uses password authentication, the proxy can behave benignly until the victim successfully logs on, and then launch the attack. The situation is much worse if the server uses Kerberos authentication (similarly, NTLM authentication), in which case the authentication happens automatically without asking the user for the password. The attack can be launched even when the victim does not intend to visit the HTTPS server: whenever the victim visits a website *http://foo.com*, e.g., a popular search engine, the proxy may insert the following invisible iframe into the webpage of *foo.com* to initiate the same attack.

```
<iframe src="https://SiteUsingKerberos.com"
style="display:none"></iframe>
```

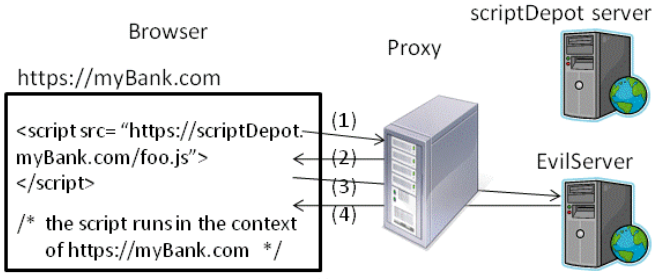
Kerberos is typically used in enterprise networks. This vulnerability allows the proxy to steal all sensitive information of the victim user stored on all HTTPS servers in the enterprise network, once the user visits an HTTP website.

B. *Redirecting Script Requests to Malicious HTTPS Websites*

After describing the PBP attacks based on the mishandling of HTTP 4xx and 5xx error messages in browsers, we now turn to another security flaw that can be exploited when browsers are dealing with HTTP 3xx redirection messages.

A benign redirection scenario is: when the user makes a request to *https://a.com*, the proxy can return a response, such as "*302 Moved temporarily. Location: https://b.com*", to redirect the browser to *https://b.com*. Similar to the previous scenario, the redirection message is in plain-text. The redirection is explicitly processed by the browser, so there is no confusion about the security context of the page – the page of the redirection target will be rendered in the context of *https://b.com*. In other words, a request redirected to *https://b.com* is equivalent to a direct request to *https://b.com*. There seems no security issue here.

However, the ability for a proxy to redirect HTTPS requests can be harmful when we consider the following scenario: many webpages import scripts from different servers. For instance, a page of *https://myBank.com* may include a script *https://scriptDepot.myBank.com/foo.js* or a third-party script *https://x.akamai.net/foo.js*. According to the HTML specification, a script element does not have its own security context but instead runs in the context of the frame that imports it. To launch an attack, a proxy may simply use a 3xx message to redirect an HTTP CONNECT request for *https://scriptDepot.myBank.com* or *https://x.akamai.net* to *https://EvilServer.com*. This will cause the script *https://EvilServer.com/foo.js* to be imported and run in the context of *https://myBank.com*. Once the script runs, it can compromise the confidentiality and authenticity of the communication in a similar manner as described previously.



(1) Browser requests https://scriptDeport.myBank.com
(2) Proxy sends HTTP 302 redirection
(3) Browser requests https://EvilServer.com
(4) Browser gets and executes foo.js from https://EvilServer.com

Figure 3. The attack using 3xx redirection message

This attack affects Firefox, Safari and Opera. IE and Chrome are immune because they only process HTTP 3xx messages after the SSL handshake succeeds. In other words, 3xx messages from the proxy are ignored by the browser for HTTPS requests.

C. *Importing Scripts into HTTPS Contexts through "HPIHSL" Pages*

Many web servers provide services of HTTP and HTTPS simultaneously. Normally, sensitive webpages, such as user login, personal identification information, and official announcement, are accessible only via HTTPS to prevent information leak and forgery. Less critical webpages are accessible via HTTP for reduced processing overhead. Webpages often need to import additional resources, such as images, scripts, and cascade style sheets. When a page is intended for HTTP, the resources are usually fetched using HTTP as well, because the page is not intended to be secure against the malicious network anyway.

However, the reality is that although less-sensitive webpages are intended to be accessed via HTTP, most of them actually can also be accessed via HTTPS. We refer to these pages as HTTP-Intended-but-HTTPS-Loadable pages, or "HPIHSL pages". When a HPIHSL page loaded in the HTTPS context imports resources using HTTP, browsers display different visual warnings: 1) IE pops up a yes/no

Table I. HTTPS domains that are compromised because HPIHSL pages import HTTP scripts or style-sheets

Compromised HTTPS domain (the domain names are obfuscated)	**The HPIHSL page that imports scripts or CSS**	**Domain and path of the HTTP script or CSS imported by the HPIHSL page**
https://www.j-store.com The checkout service is in this domain	The "men's shoes" page in *www.j-store.com*	*http://switch.atdmt.com/jaction/*
https://www.OnlineServiceX.com The checkout service is in this domain	The account help page at *www.OnlineServiceX.com/support/account*	*http://www.OnlineServiceX.com/support/ accounts/ bin/resource/*
https://www.s-store.com The checkout service is in this domain	The "Appliances" page in *www.s-store .com*	*http://content.s-store.com/js/*
https://www.CertificateAuthorityX.com A leading certificate authority	The "repository" page in *www. CertificateAuthorityX.com* imports a CSS	*http://www.CertificateAuthorityX.com /css/*
https://www.eCommerceX.com The checkout and user profiles are in this domain	The homepage of *www. eCommerceX.com*	*http://images.eCommerceX .com/media/*
https://www.sb-store.com The checkout service is in this domain	The "Furniture" page in *www.sb-store.com*	*http://graphics.sb-store.com/images/*
https://www.CreditCardX.com A credit card company	The homepage of *www.CreditCardX.com*	*http://switch.atdmt.com/jaction/COF_Homepage/v3/*
https://www.b-bank.com A bank in the Midwest	The page *www.b-bank.com/ford.asp*	*http://www.google-analytics.com/*
https://CodeRepositoryX.net, Open source projects management system. User logins are in this domain.	The homepage of *CodeRepositoryX.net*	*http://pagead2.googlesyndication.com/*
https://uboc.MortgageCompanyX.com A California mortgage company	The homepage of *uboc.MortgageCompanyX .com*	*http://uboc.MortgageCompanyX.com/Include /Utilities/ClientSide/*
https://cs.University1.edu, the department's login system is in this domain	The homepage of *cs.University1.edu*	*http://tags.University1.edu/*
https://www.eecs.University2.edu	A student's homepage *www.eecs. University2.edu/~axxxxxx*	*http://codice.shinystat.com/cgi-bin/*

dialog window. If user clicks no, the resources retrieved via HTTP will not be rendered and the lock icon will stay in the address bar. Otherwise, the resources will be rendered but the lock icon is removed; 2) Firefox pops up a warning window with an OK button. After user clicks it, the HTTP resources are rendered and a broken lock icon is displayed on the address bar. 3) Opera and Chrome automatically remove the lock icon (or or replace it with an exclamation mark) to indicate that HTTP resources have been imported.

We found that the code logic for detecting HTTP contents in HTTPS pages is triggered only when the browser needs to determine whether to invalidate/remove the lock icon on the address bar, which is only correspondent to the top-level frame of the browser. Therefore, when the top-level frame is an HTTP page, the detection is bypassed even when this HTTP page contains an HTTPS iframe that loads an HPIHSL page.

This turns out to be a fatal vulnerability for many real websites. For example, a PBP can steal the user's login information from the HTTPS checkout page of *j-Store.com* (the first row of Table I): when the user visits an HTTP merchandise page on *j-Store.com*, the proxy can insert the following invisible iframe into the page:

```
<iframe src="https://www.j-Store.com/men-shoes.html" style="display:none"> </iframe>
```

Without users' awareness, the invisible iframe loads the HPIHSL page *men-shoes.html* via HTTPS. Because this page requests a script from *http://switch.atdmt.com/jaction/* via HTTP, the proxy can provide a malicious script to serve the request. Since the script is in the inserted iframe, it will run in the context of *https://www.j-Store.com*. The PBP also overwrites the "checkout" button on the HTTP merchandise page so that when the user clicks on it, the HTTPS checkout page opens in a separate tab. The personal data entered by the user therefore can be easily obtained by the proxy's script in the invisible iframe. In addition, the proxy can impersonate the logon user to place arbitrary orders. We believe that this is a significant browser weakness: as long as any HPIHSL imports scripts or style-sheets (usually via HTTP as explained), the HTTPS domain is compromised.

To get a sense about the pervasiveness of vulnerable websites, one of the authors of this paper used HTTPS to visit HPIHSL pages for a few hours. Table I shows twelve websites that we confirmed vulnerable (the exact names of the websites are obfuscated). Each row also shows the problematic HPIHSL page and the domain of the imported script. The vulnerable websites covered a wide range of services such as online shopping, banking, credit card, open source projects management, academic information, and certificate issuance. In particular, even the homepage domain of a leading certificate authority was affected. It is a reasonable concern that many websites simultaneously opening HTTP and HTTPS ports are vulnerable.

We will discuss in Section VII how our finding is related to a paper by Jackson and Barth [7].

IV. STATIC-HTML-BASED PBP EXPLOITS

We just described a number of script-based attacks that violate the same-origin policy. By running malicious scripts in the context of victim HTTPS domains, these attacks can access or alter sensitive data that are supposed to be protected by HTTPS.

Nevertheless, in order to better understand the potential threat of PBP, thinking beyond script-based attacks is very important. Typically, for script-based security issues, the defense solutions are along the line of disabling, filtering, or guarding scripts. When a class of security problems is not always script-related, defense solutions should be explored more broadly.

In this section, we show two attacks that can be accomplished entirely by static HTML contents. They target the authentication mechanisms in browsers. In the first attack, the proxy's own page can be certified with the trusted certificate of the HTTPS server that the browser intends to communicate. In the second attack, the proxy can authenticate to the HTTPS server as a logon user.

A. Certifying a Proxy Page with a Real Certificate

In Section III.A, we have seen that the PBP proxy can supply a script in an error-response. The script will run in the HTTPS context of the victim server and compromise the confidentiality. When we reported this issue to a browser vendor, one of the vendor's proposed fixes was to disable scripts in any 4xx/5xx error-response pages, and only render static HTML contents. The proposal was based on the consideration that benign proxy error messages are valuable for troubleshooting network problems, but there is no compelling reason to allow scripts in error messages.

This fix would not block the attack that we describe below, which does not involve any script. Figure 4 illustrates how a proxy certifies a fake login page by taking advantage of a cached certificate of *https://www.paypal.com* from a previous SSL handshake. (Note that it is a browser bug. PayPal represents an arbitrary website.) IE, Opera and Chrome, but not Firefox, are vulnerable to this attack. (Note that Safari always displays the lock icon when the address bar has an HTTPS URL, even without a cached certificate, so Safari is a trivial target of the spoofing attack.)

The attack works as follows: when a browser issues a request for *https://www.paypal.com* (step 1), the proxy returns an HTTP 502 message (or any other 4xx/5xx message) that contains a `meta` element and an `img` element (step 2). The `meta` element will redirect the browser to *https://www.paypal.com* after one second. But before the redirection, the following steps happen subsequently: the `img` element requests an image from *https://www.paypal.com/a.jpg* (step 3). In order to get *a.jpg*, the browser initiates an SSL handshake with the HTTPS server. The request is permitted by the proxy at this time. After the browser receives a legitimate certificate from the HTTPS server (step 4), it will try to retrieve *a.jpg,* which may or may not exist on the server (not shown in the figure). But its existence is not important here because the purpose of the `img` element is to acquire a legitimate certificate, which has been cached in the browser now. The certificate cache is designed to enhance the performance of HTTPS by avoiding repetitive re-validation for each SSL session.

When the one-second timer is expired, the browser will be redirected to *https://www.paypal.com* (step 5). This time, the proxy returns another HTTP 502 message (or any other 4xx/5xx message) that contains a fake login page (step 6). When the browser renders this page, it picks up the cached certificate of PayPal and displays it on the address bar as if the fake page was retrieved from the real *https://www.paypal.com.*

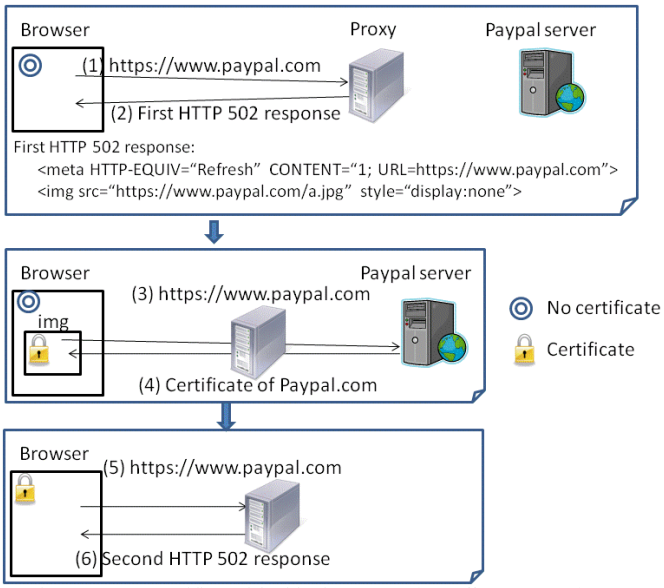


Figure 4. PBP certifies a faked login page as *https://www.paypal.com*

While the attack described here and the one described in Section III.A both take advantage of the fact that browsers render proxy's error messages in the context of HTTPS servers, these two attacks are distinguishable – In terms of the technique, this is a perfect GUI spoofing attack. Even when the user starts a fresh browser and uses a bookmark to access the HTTPS URL, he/she still gets the certified faked page. The attack is conducted in only one window and does not execute any script, therefore bypasses the pop-up blockers in today's browsers that will otherwise thwart the spoofing attack. No other attack that we describe can achieve the same result. In terms of the root cause, the proxy-page-context problem in Section III.A alone does not necessarily enable this attack, e.g., we have confirmed that Firefox is not vulnerable to the attack although it has the problem in Section III.A. A key enabler of the GUI spoofing attack is the interaction between the graphic interface and the certificate cache: for IE, Opera and Chrome, the certificate is displayed as long as it is available in the cache.

B. Stealing Authentication Cookies of HTTPS Websites by Faking HTTP Requests

The attack in Section IV.A is to impersonate a legitimate HTTPS website. We now describe an attack that allows the PBP to impersonate victim users to access HTTPS servers by stealing their cookies.

Table II. Insecure HTTPS websites due to the improper cookie protection

Website (names are obfuscated)	URL (obfuscated)	Description
StockTrader	*https://trading.StockTrader.com*	A leading stock brokerage company
eCommerceX	*https://www.eCommerceX .com*	A leading online store
V-Bank	*https:// online.vbank.com*	Online banking
ManuscriptManager	*https://mc.ManuscriptManager.com*	The submission and review system of an academic/engineering society
TravelCompany	*https://www.*TravelCompany*.com/Secure/SignIn*	A leading Internet travel company
GMail (not obfuscated as it is publicly known. See footnote 2)	*https://mail.google.com*	Google's email service
MortgageCompanyY	*https://MortgageCompanyY.com*	Mortgage lender
UtilityCompanyX	*https://www.UtilityCompanyX.com*	A utility company in the west coast of the United States.
GovermentServiceX	*https://egov.GovermentServiceX.gov*	A web service for U.S. immigration cases

Cookies are pieces of text that browsers receive from web servers and store locally. They are used to maintain the states of HTTP transactions, such as items in consumer's shopping carts and personalized settings of user webpages. In addition, they are used as an important mechanism for web servers to authenticate individual users. After a user successfully logs on a server, the server sends some cookies to be stored in the user's browser, which uniquely identify the session between the server and the user. Next time when the user accesses the server, these cookies are presented to the server as a proof of the identity of the user.

Browsers use the same-origin policy to determine whether cookies can be attached to requests or accessed by scripts. The policy specifies that: 1) Cookies of a domain can only be attached to the requests to the same domain; 2) Cookies of a domain are only accessible to scripts that run in the context of the same domain. However, unlike the same-origin policy of script and DOM, the same-origin policy of cookies does not make a distinction between HTTP and HTTPS by default. In the default scenario, cookies of *http://a.com* may be accessed by pages or scripts of *https://a.com*, and vice versa. Optionally, a SECURE attribute [5] can be set to ensure that cookies can only be read by pages in the HTTPS context and be attached to the HTTPS requests (of course, after the SSL handshake).

We found that many websites do not set the SECURE attribute for cookies that identify HTTPS sessions[2]. As an example, an author of the paper investigated about 30 websites in which he owns an account. About one-third of the websites used cookies for authentication but did not set the SECURE attribute for them. Every website was verified individually to show that the stolen cookie was sufficient to allow the attacker to get into the logon session from an arbitrary machine and to perform arbitrary operations on behalf of the victim user. These nine websites are listed in Table II, with their names and URLs obfuscated. They cover a wide range of services such as stock broker, online shopping, online banking, academic paper reviewing, email service, mortgage payment, utility billing, government service, and traveling. They affect many different aspects of a person's online security.

It is straightforward to launch the attack: the proxy waits until the user logs into the server (usually after seeing a few CONNECT requests), e.g., the stock trading website *https://trading.StockTrader.com*. After that, once the browser requests any HTTP page (including a page requested from another browser tab or any tool bar), the proxy embeds an iframe of *http://trading.StockTrader.com* in the HTTP response. When the browser renders the iframe, it makes an HTTP request for *http://trading.StockTrader.com*, exposing the authentication cookie in plain text to the proxy.

Given that a significant fraction (one-third) of the HTTPS websites that we examined have this problem and many of them are reputable, we believe this vulnerability exists in many other HTTPS websites as well. Although it is possible that inexperienced developers do not have knowledge about the SECURE attribute of cookies, the fact that reputable websites also make this mistake suggests that the concept of the SECURE attribute is commonly misconceived. The SECURE attribute is often vaguely defined as a mechanism to prevent malicious HTTP pages. It is never made clear that when the network is assumed untrusted, the SECURE attribute should be considered as a mechanism to prevent malicious proxies and routers. Without this clear interpretation, a developer might have a misconception: my HTTP pages are very secure (or "my website does not run HTTP at all"). Why bother to prevent my own HTTP pages from stealing cookies of the HTTPS sessions on my website?

V. Feasibility of Exploitations in Real-World Network Environments

By definition, the security of communications over HTTPS should not rely on the integrity of any intermediate node in network path, such as proxies and routers. As

[2] We discovered this issue in June 2007, and reported it privately in July 2007. Mike Perry independently discovered the Gmail vulnerability due to this issue, and posted a description on *SecurityFocus.com* on August 6th, 2007 [12]. According to our private email communication with Nick Weaver, Nick also conceived this attack scenario independently in the time frame.

described in the previous sections, however, the guarantees of HTTPS can be subverted when a malicious or compromised proxy is being used. There are many circumstances where proxies are commonly used and therefore the PBP vulnerabilities can be easily exploited: (1) Mobile environments such as conference rooms, airports, hotels and hospitals [22]; (2) Corporate and university networks, e.g., Microsoft's corporate network and the campus networks in Berkeley and UCSD [23]; (3) Free third-party proxies on the Internet [24]. In these cases, proxies may be used for various legitimate reasons, such as billing, traffic regulation, and traffic anonymization. However, if they are infected by viruses, hijacked by attackers, or configured by malicious insiders, the PBP attacks can be launched.

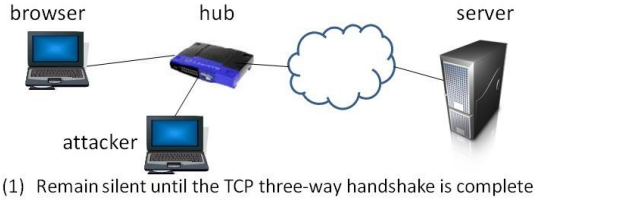
In this section, we will show that in real-world network environments, the PBP vulnerabilities can be exploited more easily than hacking into the proxy machine. An attacker can exploit the vulnerabilities even when the victim is not knowingly using an untrusted proxy. The attacker only needs the capability of sniffing users' traffic and sending fake packets back to browsers. An attacker can easily do this by sitting in the vicinity of victim users in a wireless environment or connecting to the same local area network (LAN) of victim users in a wired environment. Note that the attack scenarios to be described do not show any additional vulnerability, but demonstrate that the PBP vulnerabilities described earlier result in serious consequences for people's online security.

The tactic of our attacks is to impersonate a legitimate proxy or insert an unwanted proxy into the communication path without the user's awareness. We will discuss a few basic elements in this tactic: (1) *TCP hijacking* – It is a known fact that anyone who can sniff IP packets can hijack the TCP connections, and thus impersonate clients and servers; (2) *Proxy-Auto-Config (PAC) mechanism* [20] – Alternative to manual configuration, browsers use the PAC mechanism to obtain a script from a server and configure proxy settings by the script; (3) *Web-Proxy-Auto-Discovery protocol (WPAD)* [21] – All browsers support WPAD. WPAD makes the proxy configuration completely under the hood: it attempts to discover a proxy, and automatically falls back to the "no-proxy" setting if the attempt fails. Using WPAD, the same browser machine can access the web at office, hotel and home without changing any setting. Since TCP, PAC and WPAD are not cryptographic protocols, they are not expected to be resilient against an attacker who can access the network traffic. However, combining these facts with PBP vulnerabilities, HTTPS' properties become very easy to break in reality.

We have built Ethernet and wireless testbeds to show various attack scenarios. The details are provided in the following subsections.

A. A Short Tutorial of TCP Hijacking

It is well known that an attacker who can sniff TCP traffic can impersonate the sender or the receiver of a TCP connection. This technique is referred to as TCP hijacking and is shown in Figure 5. The attacker is at a location where he can sniff the TCP packets between the browser machine and the server. To simplify description, we assume that the attacker connects to the same Ethernet hub as the user machine. When the browser tries to establish a TCP connection with the server, the attacker does nothing but wait for the completion of TCP three-way handshake. When the attacker receives the packet which contains an HTTP request sent by the browser, it parses the packet to extract the IP addresses, the port numbers, and the current sequence numbers of both the browser and the server. It then uses this information to fake a server response packet with appropriate IP and TCP headers and payload data. The fake packet is immediately sent back to the browser.



(1) Remain silent until the TCP three-way handshake is complete
(2) Sniff the browser's packet, get the TCP sequence number
(3) Construct a faked packet header containing the IP addresses, the port numbers and the sequence numbers from the sniffed packet.
(4) Add arbitrary payload data. Send the packet to the browser immediately.

Figure 5. A typical TCP hijacking

Since the attacker can only sniff but not intercept packets, the server will still return a legitimate response packet to the browser. Given that the distance between the attacker and the browser is equal or shorter than the distance between the browser and the server, the fake response packet generated by the attacker almost always arrives before the legitimate response packet from the server. Although this is a race between the attacker and the server, because the attacker has already prepared most of the faked response and only waits to fill in a few header fields, it is easy to win the race. The browser will accept the fake packet and discard the legitimate packet as a duplicate, because both packets have the same TCP sequence number.

B. PBP Exploits by a Sniffing Machine

As we stated earlier, connecting to a proxy is necessary in many circumstances, such as corporate networks, hotels, and conferences, for the purpose of billing or auditing [22][23][24]. Browsers' proxy settings can be configured manually, or by specifying the URL of the PAC script, or by WPAD. The attacker has several options accordingly. Figure 6 shows the user interface of proxy settings for IE and Chrome. Other browsers' user interfaces are almost functionally identical.

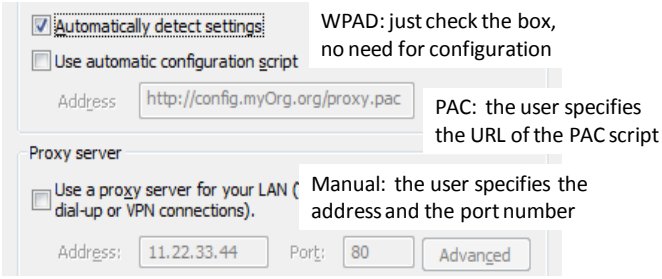


Figure 6. Proxy setting options for IE and Chrome

Browsers with manual proxy-settings. Manual configuration requires the (advanced) user to enter the hostname/IP address and the port number of the specific proxy server. The attacker needs to hijack the TCP connection between the browser and the proxy to impersonate the proxy.

Browsers configured by PAC scripts [20]. A browser can fetch a PAC (Proxy Auto-Config) script from a server by specifying the URL, such as *http://config.myOrg.org /proxy.pac*. The script contains a special function `FindProxyForURL(url,host)`, which returns a string containing one or more proxy specifications given a URL and a hostname. In practice, proxy settings are normally cached for better performance. To attack this browser, the attacker can hijack the TCP connection to impersonate the PAC server *config.myOrg.org*. The following PAC script is served to the browser. The browser will use "proxy.evil.com:80" as its proxy.

```
function FindProxyForURL(url,host)
  {return "PROXY proxy.evil.com:80";}
```

The advantage of impersonating the PAC server, compared to impersonating the proxy server, is that the hijacking only needs to be done once and the browser's proxy setting will be changed permanently.

Browsers enabling WPAD [21]. WPAD (Web Proxy Auto Discovery) is the only option for users to browse the web from different networks without changing the configuration. When WPAD is enabled, the browser does not initially know the URL of the PAC script, but asks the DHCP server for it. If DHCP server does not have the information or does not respond, the browser asks the DNS servers. Once the URL of the PAC script is obtained, the browser fetches the script and configures its proxy settings. If the browser cannot find any proxy configuration script, it automatically falls back to the "no-proxy" state, in which the browser does not access the web through any proxy. Our attack program sniffs browser packets. When there is a WPAD query for DHCP or DNS server, the program replies immediately with the URL of a malicious PAC script on the attacker machine.

Home networks typically have no HTTP proxy servers, so it may be an expectation that online banking at home is secure. It is worth noting that whether there is a proxy in a home network does not affect the security. The security is only affected by whether the browser has any one of the proxy settings enabled. For example, if a laptop sets the WPAD capability in the office hours in a corporate network, it will be insecure to do online banking at home in the evening with the proxy setting unchanged, because the attacker can fake a WPAD response to convince the browser that there is a proxy.

Browsers with proxy settings disabled. If a user does disable proxy service in browsers, the vulnerabilities described in Section III.A, III.B and IV.A are no longer exploitable because the browser will directly establish HTTPS connections with servers instead of tunneling the connections through proxies, evading the code paths that trigger the vulnerabilities. However, the remaining two vulnerabilities described in Section III.C and IV.B can be exploited as they only require attackers to sniff HTTP requests and forge HTTP responses.

The default proxy settings of the browsers. If a user has never modified any proxy settings since the installation of the browser, the default settings vary in different browsers: (1) Firefox does not enable any proxy setting by default; (2) IE enables and uses WPAD in its very first run after the installation. If this first use is successful, the WPAD setting is checked, otherwise it is unchecked. Chrome always uses IE's setting; (3) After the installation, Opera's initial setting is the same as IE's setting.

Therefore, even in a fresh IE, Opera or Chrome at home, the proxy setting will be enabled if the attacker responds to all WPAD requests that he/she receives.

C. Attack Implementations

We implemented all attack scenarios in both the Ethernet and wireless environment. In the Ethernet, we used WinPcap [18] to sniff and inject packets in Windows platform. WinPcap is a network monitoring library; it provides a set of APIs which allow us to capture all raw packets received by network interface card (NIC) and send raw packets through NIC. These raw packets include link layer headers, IP headers, TCP headers, and full payload data. While a NIC normally discards packets whose physical (MAC) addresses do not match that of the NIC, WinPcap can set the NIC in the promiscuous mode such that all packets received by the NIC will be passed up.

Wireless environments are more dangerous. Given the nature of wireless networks, attackers can sniff wireless packets in the air when they are in the vicinity of the wireless access points which victims are using (unless per-user encryption schemes WPA and WPA2 are deployed, which will be discussed in Section VI.B). Conceptually, the attacks in a wireless network are the same as that in an Ethernet LAN. However, we need to resolve a number of implementation issues to enable the attacks, which are described below.

Table III. Vulnerability reporting and browser vendors' responses

	Microsoft (IE)	Mozilla (Firefox)	Apple (Safari)	Opera	Google (Chrome)
Vulnerability in Section III.A	Fixed in IE8	Fixed in version 3.0.10	Fixed before version 3.2.2	Fixed in Dec.2007	Fixed in version 1.0.154.53
Vulnerability in Section III.B	N/A	Fixed in version 3.0.10	Fixed before version 3.2.2	Fixed in Dec. 2007	N/A
Vulnerability in Section III.C	Suggest fix for the next version	Vulnerability acknowledged, Fix proposed	Vulnerability acknowledged	Vulnerability acknowledged	Fix planned
Vulnerability in Section IV.A	Fixed in IE8	N/A	Fixed before version 3.2.2	N/A	Fixed in version 1.0.154.53
Vulnerability in Section IV.B	Not reported (Non-browser issue)	Not reported (Non-browser issue)	Not reported (Non-browser issue)	Not reported (Non-browser issue)	Not reported (Non-browser issue)

Although WinPcap works well on most Ethernet NICs, it is not properly supported by many wireless NICs. First, many wireless NICs do not support the promiscuous mode for power conservation. Second, WinPcap device driver assumes Ethernet as its default link layer, which is incompatible with most wireless NICs. However, we do find that certain wireless NICs (e.g. Dell TrueMobile 1300 WLAN Mini-PCI Card) work with WinPcap and support the promiscuous mode. On these NICs, WinPcap emulates an Ethernet layer by automatically creating fake Ethernet frames from WiFi frames. In addition, we have developed a specific packet sniffer/injector that works with D-Link AG-132 Wireless USB Adapter in Windows platform.

VI. MITIGATIONS AND FIXES

In this section, we describe how browsers vendors fixed or planned to fix the vulnerabilities reported in this paper. We also discuss possible ways to mitigate the impact of the class of PBP vulnerabilities before they are discovered and patched.

A. Fixes of the Vulnerabilities

We have reported these vulnerabilities (except the authentication cookie vulnerability) to the affected browser vendors: Microsoft's IE team, Mozilla's Firefox team, Opera Software and Google's Chrome team. Since the authentication cookie vulnerability in Section VI.B is due to improper setting of cookie attribute by individual websites, we have to inform the websites instead of the browser vendors. Table III shows the browser vendors' responses to each of these vulnerabilities. The vendors have acknowledged the vulnerabilities reported by us. The vulnerabilities described in Section III.A and III.B have been addressed by all vendors: Microsoft has fixed them in IE8. Firefox, Opera, Safari and Chrome have also fixed them in their latest versions.

IE8, Firefox, Safari, Opera and Chrome fixed the vulnerability in Section III.A by displaying a local error page when receiving a 4xx/5xx response before the SSL handshake succeeds. Opera and Safari fixed the vulnerability in Section III.B by ignoring the proxy's 3xx redirections. As a proposal for the vulnerability in Section III.C, Mozilla plans to fix it by blocking any script/CSS resources imported by HTTP into HTTPS context, or reliably display a warning. Microsoft and Opera are considering a "defense-in-depth" patch, of which the details have not been confirmed.

Browser vendors are not in the best position to fix the authentication cookie vulnerability described in Section IV.B, because currently there is no mechanism to for the browsers to know if a cookie value is for the authentication purpose or meant to be shared with the corresponding HTTP domain. The cookie's secure attribute largely depend on the application semantics.

B. Mitigations by Securing the Network

Because HTTPS is designed fundamentally for secure communications over an insecure network, it is of course an unconvincing "solution" that we secure the network in order to secure HTTPS. However, in practice, network-based mitigation approaches are still valuable to consider because it is not safe to assume every machine will be fully patched. More importantly, we believe there will be future vulnerabilities similar to what we have discovered. Good mitigations that are effective against known attacks may mitigate future/unknown attacks.

At the high level, users should be cautious about plugging their machines into untrusted network ports, connecting to unknown wireless access points (APs), or using arbitrary network proxies. In cases when users must go through them to access the network, they should avoid using the network for critical transactions, such as online banking. In enterprise networks where Kerberos authentication is being used, network administrators should prevent any unauthorized sniffing of user traffic.

Since the PBP attacks are so easy to launch if the attacker has the ability to intercept or sniff traffic content, a straightforward mitigation approach is to encrypt the content transmitted on the network. Fortunately, there have already existing techniques that are applicable in different scenarios:

- Almost all wireless APs support encryption, which make it difficult for adversaries to sniff traffic in the air. Among the commonly available encryption schemes, WPA and WPA2 are more secure, because they maintain per-user keys. WEP uses a static shared key

among all the users who connect to the same AP. It is widely known that WEP is easy to break [1].

- Sometimes, users must rely on untrusted networks to access the Internet, e.g., in hotels, airports, conferences, and coffee shops. They may secure their traffic by using secure Virtual Private Network (VPN) if such option is available. Secure VPN allows a client to establish a secure connection with a VPN server, in which case all the traffic between them is encrypted. Once the connection is established, all requests and replies will be tunneled through the VPN server. Conceptually the users' machines are connected to the enterprise network of the VPN server.
- Enterprise networks should deploy IPSec to encrypt traffic at the IP-layer. Today, IPSec coexists with regular IP in enterprise networks. There are lots of opportunities for PBP attackers to intercept or sniff users' traffic in enterprise networks. For example, large enterprise networks typically have thousands of network ports. Attackers can easily plug in their own wireless APs to a network port without being detected for a long time. These APs are often referred to as Rogue APs and allow attackers to gain unauthorized access to enterprise network (e.g., sniffing users' traffic). Another example is Network Load Balancing (NLB) where servers in the same load balancing group share a broadcast address [10]. This facilitates packet sniffing. To resolve these issues, it is important that all hosts involved in the communication must use IPSec to protect users from PBP attacks. To understand its importance, let us assume that IPSec is only used by all the proxy servers but not the PAC servers. Adversaries may still intercept/sniff the requests to PAC servers and feed malicious PAC scripts to browsers as we described earlier. Similarly, IPSec must be deployed on other types of servers that provide basic network services, such as DHCP servers and DNS servers.

VII. Related Work

Violations of the same-origin policy are one of the most significant classes of security vulnerabilities on the web. Classic examples include cross-site scripting (aka, XSS) and browser's domain-isolation bugs: (1) XSS is commonly considered as a web application bug. Vulnerable web applications fail to perform sanity checks for user input data, but erroneously interpret the data as scripts in the web application's own security. Many researchers have proposed techniques to address XSS bugs. A compiler technique is proposed by Livshits and Lam to find XSS bugs in Java applications [9]. Based on the observation that XSS attacks require user-input data be executed at runtime, Xu et al proposed using taint tracking to detect the attacks [19]. There are many other research efforts in the area of XSS that we cannot cite due to space constraints. (2) Historically all browser products had bugs in their domain-isolation mechanisms, which allow a frame tagged as *evil.com* to access the document in another frame tagged as *victim.com* on the browsers. Security vulnerability databases, including *SecurityFocus.com*, have posted many bugs against IE, Firefox, Opera, etc. These vulnerabilities are discussed in [4].

The contribution of this paper is not to show that HTTPS is breakable. In fact, people already understand that HTTPS security is contingent upon the security of clients, servers and certificate authorities. Binary-executable-level threats, such as buffer overruns, virus infections and incautious installations of unsigned software from the Internet, allow malicious binary code to jump out of the browser sandbox. In particular, when a malicious proxy or router is on the communication path, the binary-level vulnerabilities can be exploited even when the browser visits legitimate websites. Unsurprisingly, once the browser's executable is contaminated, HTTPS becomes ineffective. In addition to the binary-level vulnerabilities, XSS bugs and browser's domain-isolation failures may compromise HTTPS. Furthermore, some certificate authorities use questionable practices in certificate issuance, undermining the effectiveness of HTTPS. For example, despite the know weaknesses of MD5, some certificate authorities have not completely discontinued the issuance of MD5-based certificates. Sotirov, Stevens, et al have recently shown the practical threat of the MD5 collision by creating a rogue certificate authority certificate [15]. In contrast to these known weaknesses, the contribution of our work is to emphasize that the high-level browser modules, such as the HTML engine and the scripting engine, is not thoroughly examined against the PBP adversary, and PBP indeed has its uniqueness in attacking the end-to-end security.

HTTPS has usability problems because of its unfriendliness to average users. Usability studies have shown that most average users do not check the lock icon when they access HTTPS websites [14]. They are willing to ignore any security warning dialog, including the warning of certificate errors. Logic bugs in browsers' GUI implementations can also affect HTTPS effectiveness. In [3], we show a number of logic bugs that allow an arbitrary page to appear with a spoofed address and an SSL certificate on the address bar.

The HPIHSL vulnerability described in Section III.C is related to the "mixed content" vulnerability in [7] by Jackson and Barth. Jackson/Barth and we exchanged the early drafts of [7] and this paper in October 2007 to understand the findings made by both parties, which are distinguishable in the following aspects: (1) the scenario in [7] is that the developer of an HTTPS page accidentally embeds a script, an SWF movie or a Java applet using HTTP, while our main perspective is about loading an HTTP-intended page through HTTPS; (2) we discover that

the warning message about an HTTP script in an HTTPS frame can be suppressed by placing the HTTPS frame in a HTTP top-level window, while [7] argues that such a warning is often ignored by users; (3) we found twelve concrete e-commerce and e-service sites that we sampled where the vulnerability based on HPIHSL pages exists. This suggests that this vulnerability may currently be pervasive. In [7], there is no argument about the pervasiveness of the accidental HTTP-embedding mistakes made by developers.

Karlof, Shankar, et al envision an attack called "dynamic pharming" to attack HTTPS sessions by a third-party website. The attack is based on the assumption that the victim user accepts a faked certificate [8]. Because HTTPS security crucially relies on valid certificates, accepting a faked certificate is a sufficient condition to void HTTPS guarantees. To address dynamic pharming, the authors propose locked same-origin-policies to enhance the current same-origin-policy. These policies do not cover PBP attacks discussed in Sections III.B, III.C, IV.A and IV.B. For the attack in Section III.A, if developers understand that 4xx/5xx pages from the proxy cannot bear the context of the target server, then the current same-origin-policy is already secure; if they overlook this, as all browser vendors did, it is unlikely that the mistake can be avoided in the implementations of the locked same-origin-policies.

Researchers have found vulnerabilities in DNS and WPAD protocol implementations. Kaminsky showed the practicality of the DNS cache poisoning attack, which can effectively redirect the victim machine's traffic to an IP address specified by the attacker [17]. This attack can be used to fool the user to connect to a malicious proxy. Researchers also found security issues about WPAD, e.g., registering a hostname "wpad" in various levels of the DNS hierarchies can result in the retrievals of PAC scripts from problematic or insecure PAC servers [2][16]. Unlike these findings, our work does not attempt to show any vulnerability in WPAD. It is unsurprising that the communication over an unencrypted channel is insecure when the attacker can sniff and send packets on the network – several possibilities of maliciously configuring the browser's proxy settings were documented in [11]. We discuss PAC, WPAD and the manual proxy setting only as a feasibility argument of the PBP vulnerabilities.

VIII. Conclusions and Future Work

The PBP adversary is a malicious proxy targeting browsers' rendering modules above the HTTP/HTTPS layer (e.g., the HTML engine) in order to void the HTTPS' end-to-end security properties. The specific attack strategy of the PBP is to feed malicious contents into the rendering modules through the unencrypted channel, then access secret data (or forge authentic data) above the cryptography of HTTPS. We emphasize that this adversary must be carefully examined so that the security guarantees ensured by HTTPS are preserved in the whole system. The vulnerabilities discussed in the paper exist across all major browsers and a wide range of websites, indicating that they are not simply due to accidental mistakes in implementations, but due to the unawareness of the threat of the PBP adversary in the industry.

We evaluated the feasibility and the consequences of the discovered vulnerabilities in realistic network settings. While the PBP attacks can be launched against users who rely on untrusted proxy to access the Internet, it is certainly not the only circumstance where users become vulnerable. We conducted experiments in both wired and wireless testbeds to demonstrate that the PBP attacks can be launched in many real-world scenarios, such as public wireless hotspots, Internet access in hotels, enterprise networks, and sometimes even home networks. Specifically, for a browser that has its proxy capability enabled, an attacker who can sniff the browser's raw traffic can accomplish all the attacks.

We consider our findings as an initial step towards a comprehensive understanding of secure deployments of HTTPS on the web. Because of the existing complexity and the rapid development of web technologies, we believe that the security community needs bigger efforts to investigate in this new problem space. What we discovered so far by no means cover all possibilities of PBP attacks. We provided a set of network measures to help mitigate PBP attacks.

Beyond HTTPS, it is also necessary to use holistic thinking and evaluation to ensure secure deployments of other cryptographic protocols, such as IPSec and Kerberos. By definition, IPSec provides IP layer authentication and/or encryption, and Kerberos enables domain-user authentications over an untrusted network. Many websites on enterprise networks run HTTP over IPSec, using the Kerberos authentication. What does this architecture mean in terms of security, if browsers' proxy configuration traffic is unencrypted? We believe the research in this general area will have significant practical relevance – cryptographic protocols are rigorously-designed foundations for secure communications, which can effectively protect users only if the protocol deployments in real systems are secure against the same adversary that the protocols try to defeat.

Acknowledgements

We thank Dan Simon, Eric Lawrence and David Ross for offering their insightful discussions about cryptography, proxy and browser security topics. Our communications with Adam Barth and Collin Jackson helped improve the paper. Kieron Shorrock, Mike Reavey and Andrew Cushman of Microsoft Security Response Center spent significant efforts helping us resolve various issues around this research. We thank anonymous reviewers and our shepherd Bill Aiello for helpful and actionable comments.

REFERENCES

[1] Andrea Bittau, Mark Handley, Joshua Lackey, "The Final Nail in WEP's Coffin," the 2006 IEEE Symposium on Security and Privacy, Oakland, CA

[2] Grant Bugher. "WPAD: Internet Explorer's Worst Feature," http://perimetergrid.com/wp/2008/01/11/wpad-internet-explorers-worst-feature/

[3] Shuo Chen, Jose Meseguer, Ralf Sasse, Helen J. Wang, Yi-Min Wang, "A Systematic Approach to Uncover Security Flaws in GUI Logic," in IEEE Symposium on Security and Privacy, Oakland, California, May 2007.

[4] Shuo Chen, David Ross, Yi-Min Wang, "An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism," in ACM Conference on Computer and Communications Security, Alexandria, VA, Oct-Nov 2007.

[5] Cookie Property. MSDN. http://msdn2.microsoft.com /en-us/library/ms533693.aspx

[6] T. Dierks and E. Rescorla. RFC5246: The Transport Layer Security (TLS) Protocol. http://tools.ietf.org /html/rfc5246

[7] Collin Jackson and Adam Barth, "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks," in Proceedings of the 17th International World Wide Web Conference (WWW2008)

[8] Chris Karlof, Umesh Shankar, J.D. Tygar, and David Wagner, "Dynamic Pharming Attacks and Locked Same-origin Policies for Web Browsers," the Fourteenth ACM Conference on Computer and Communications Security (CCS 2007), November 2007.

[9] Benjamin Livshits and Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis," in Proc. Usenix Security Symposium, Baltimore, Maryland, August 2005

[10] Network Load Balancing: Frequently Asked Questions for Windows 2000 and Windows Server 2003. http://technet2.microsoft.com/windowsserver/en/library/884c727d-6083-4265-ac1d-b5e66b68281a1033.mspx? mfr=true

[11] Andreas Pashalidis. "A Cautionary Note on Automatic Proxy Configuration," Proceedings of the IASTED International Conference on Communication, Network, and Information Security (CNIS 2003).

[12] Mike Perry. "Active Gmail "Sidejacking" - https is NOT ENOUGH," http://www.securityfocus.com/archive/1/475658/30/0/threaded

[13] Jason Rafail, "Cross-site scripting vulnerabilities," www.cert.org/archive/pdf/cross_site_scripting.pdf

[14] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer, "The Emperor's New Security Indicators: An evaluation of website authentication and the effect of role playing on usability studies," in 2007 IEEE Symposium on Security and Privacy, Oakland, CA. May 20-23, 2007.

[15] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger. "MD5 considered harmful today -- Creating a rogue CA certificate," http://www.win.tue.nl/hashclash/rogue-ca/#sec71

[16] Niels Teusink. "Hacking random clients using WPAD," http://blog.teusink.net/2008/11/about-two-weeks-ago-i-registered-wpad.html

[17] US-CERT. "Multiple DNS implementations vulnerable to cache poisoning," http://www.kb.cert.org/vuls/id/800113

[18] WinPcap: The Windows Packet Capture Library. http://www.winpcap.org/

[19] Wei Xu, Sandeep Bhatkar and R. Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in Proc. the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 2006.

[20] MSDN Online. Using Automatic Configuration, Automatic Proxy, and Automatic Detection. http://www.microsoft.com/technet/prodtechnol/ie/reskit/6/part6/c26ie6rk.mspx?mfr=true

[21] MSDN Online. WinHTTP AutoProxy Support. http://msdn.microsoft.com/en-us/library/aa384240.aspx

[22] Internet access configuration instructions for some conferences, hotels, hospitals and airports that require proxies.
http://homepage.eircom.net/~acsi/encs08.htm
http://www.hw.ac.uk/uics/Help_FAQs/WiFi_FAQs.html
http://homepage.eircom.net/~acsi/encs08.htm
http://www.ucd.ie/mcri/resources/IP_logistics_students.pdf
http://www.vistagate.com/Demo/Administration/Help.htm
http://www.grh.org/patWireless.html

[23] Internet access configuration instructions for some university departments and libraries that require proxies.
http://groups.haas.berkeley.edu/hcs/howdoi/AirBearsXP.pdf
http://www.lib.berkeley.edu/Help/proxy_setup_ie5-7_dialup.html
http://meded.ucsd.edu/edcom/tech_support/remote_access/web_proxy/internet_explorer/
http://physics.ucsd.edu/~sps/html/resources/articles/sciamsetup.html
http://www.lib.ucdavis.edu/ul/services/connect/proxy/step1/iewindowslong.php

[24] Anonymizer: free web proxy, CGI proxy list, free anonymizers and the list of web anonymizers list.
http://www.freeproxy.ru/en/free_proxy/cgi-proxy.htm