

Man-in-the-Browser-Cache: Persisting HTTPS Attacks via Browser Cache Poisoning

Paper under submission. Please do not distribute.

Yaoqi Jia
National University of
Singapore
jiayaoqi@comp.nus.edu.sg

Prateek Saxena
National University of
Singapore
prateeks@comp.nus.edu.sg

Yue Chen
Beihang University
chenyue@ee.buaa.edu.cn

Jian Mao
Beihang University
maojian@buaa.edu.cn

Xinshu Dong
Advanced Digital Sciences
Center
xinshu.dong@adsc.com.sg

Zhenkai Liang
National University of
Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

In this paper, we present a systematic study of *browser cache poisoning (BCP)* attacks, wherein a network attacker performs a one-time Man-In-The-Middle (MITM) attack on a user's HTTPS session, and substitutes long-lived malicious resources for the original ones. BCP attacks require users to click through one SSL warning on any site, after which the poisoned resources are persistently cached, affecting the user's future HTTPS sessions in that web browser. We investigate the feasibility of such attacks on five mainstream desktop browsers (e.g., Chrome, Firefox, etc.) and 16 popular mobile browsers. We find that browsers are highly inconsistent in their caching policies for loading resources over broken SSL connections. In particular, the majority of desktop browsers (99% of the market share) and mobile browsers (over a billion user downloads) are affected to BCP attacks to a large extent. Existing solutions for safeguarding HTTPS connections fail to provide comprehensive defenses against this threat. We provide guidelines for users and browser vendors to defeat BCP attacks, and propose defense techniques for website developers to mitigate an important subset of BCP attacks on the existing deployed base of browsers. We have reported our findings to the related browser vendors. For Chrome, Google has confirmed our reported vulnerability in HTML5 AppCache and is deploying a fix.

1. INTRODUCTION

In today's browsers, an HTTPS session can be rendered insecure if a user Alice clicks through the browser's SSL warnings under a Man-in-the-middle (MITM) attack. Recent works have measured the click-through rates for SSL warnings in the field [15, 26, 54] indicating that 50% or more users click through SSL warnings, and investigated ways for improving warnings [31, 54]. Even worse, users sometimes may not have a choice but to click through the warnings, e.g., when in a hotel WiFi with a proxy that intercepts web sessions [23]. In this paper, we study the extent to which Alice's future HTTPS sessions are rendered insecure if she clicks through one SSL warning. Our experiments find that a single unsuspecting click-through on the targeted site A can compromise all of Alice's future HTTPS sessions with site A, and worse — even with sites other than A she visits in the same browser. These attacks persist over time and across websites because of browsers' caching of resources loaded over broken SSL connections. In addition, the policies for browser caching on SSL errors are implemented differ-

ently in different browsers opening a large fraction of mobile and desktop users to such threats.

We study *browser cache poisoning (BCP)* attacks, wherein network attackers intercept connections and substitute malicious resources for the original ones in the targeted site A over HTTPS, if Alice clicks through one SSL warning on any site. By setting long-lived cache headers, the malicious copies will be cached in Alice's browser for a long time, and these poisoned resources persistently compromise all future sessions using that browser even over correct HTTPS. We classify BCP attack vectors into three types: same-origin, cross-origin and extension-assisted. For same-origin BCP attacks on an origin A, the attacker poisons A's pages once, and persists them over time using the browser's cache. Attackers can mount cross-origin BCP attacks by corrupting subresources imported by the target site A from a different domain B. Finally, subresources injected by browser extensions¹ can be used to affect all sessions originating from a browser. Poisoned resources in web cache² are shared across all sites, irrespective of whether they are intercepted on a different site or the same one. The implication of these attacks is that if Alice clicks through one SSL warning on any site, all her future sessions on that browser may be compromised until she clears the cache.

Browsers vary substantially on how they display warnings about broken HTTPS connections and in their caching policies. We evaluate browser cache poisoning attacks on five mainstream desktop browsers (e.g., Firefox and Chrome) and 16 popular mobile browsers (e.g., Android Default Browser and Dolphin), which cover over 99% desktop browser users³ [13] and more than one billion mobile browser users (by download statistics). For SSL warnings, we find that CM browser that has 10 million users does not check the validity of sites' certificates and never shows SSL warnings. In Firefox 3.6, Internet Explorer 8, and other old version browsers, SSL warnings can be hidden/overlaid in frames using clickjacking techniques. Further, a majority of mobile browsers prompt users with incomplete information in SSL warnings, making it difficult for security-conscious users to make informed decisions. For example, when users visit sites with wrong certificates, several SSL

¹Extensions includes extensions in Chrome and Safari, and add-ons in Firefox and Opera.

²In this paper, we use web cache to refer to the default browser cache, which caches all HTTP/HTTPS resources unless the `no-cache` header is set.

³The other desktop browsers have less than 1% market share.

warnings do not contain the site’s name and certificate’s content, or contain the name of the top-level URL rather than the hijacked site’s actual URL. Browser extensions in Safari and Opera can inject HTTP scripts into the HTTPS site without raising any SSL or mixed-content warnings, unlike in Chrome and Firefox. Inconsistencies of SSL warnings affect Alice’s decision to click through on certain sites, opening up the opportunity for BCP attacks.

Browsers provide more than one kind of cache, e.g., web cache and HTML5 AppCache, and enforce different caching policies. For correct HTTPS connections, all browsers respect the header’s directives and cache resources properly. However, for click-through HTTPS connections, all 20 browsers but Safari (desktop version) cache resources in web cache or in HTML5 AppCache. Caching policies for the HTML5 AppCache are different between Chrome and Safari. Although Safari has a proper caching policy for HTTPS, it is still vulnerable to such attacks via extensions which do not raise SSL warnings if they inject HTTP scripts into HTTPS sites. We discuss these differences in depth in Section 3, and show that all five desktop browsers and 16 mobile browsers are susceptible to browser cache poisoning attacks. We have reported our findings to these browser vendors. For Chrome, Google has confirmed our reported bug in AppCache and is deploying a fix by not caching resources over broken HTTPS in AppCache as we suggest in Section 5.

Many existing defenses against HTTPS MITM attacks, e.g., Channel IDs [16], SISCAs [45], HSTS [36], HPKP [28], and other best practices (e.g., enabling CSP [3]) are witnessing real-world adoption. However, none of them provide a comprehensive defense, although these incidentally protect against a subset of BCP attacks. We study the Alexa Top 100 websites, and find that only five sites, such as *facebook.com* and *twitter.com*, are protected in some browsers due to these defenses. By analyzing 31,377 HTTPS websites out of Alexa Top 1,000,000 sites, we find that only 510 (1.63%) sites enable the proper protection. Therefore, developers should not rely on these defenses as a panacea for BCP attacks. We discuss how browser vendors can make their browser caching policies consistent and correct, thereby eliminating the threat. We also provide guidelines for users, and propose defense techniques for web developers to mitigate the impact of these attacks on the existing deployment of browsers.

Contributions. BCP attacks have been conceptually discussed in previous works [47, 52, 55]. In this work, we make the following contributions:

- **Susceptibility of desktop and mobile browsers.** Through experiments, we find the inconsistency of SSL warnings that can mislead victims to click through warnings and the incoherence of browser caching policies that makes browsers vulnerable to BCP attacks. We find that five mainstream desktop browsers and 16 popular mobile browsers are susceptible to such attacks, and 99% desktop browser users and over one billion mobile users are affected. Meanwhile, only five sites of Alexa Top 100 and 1.63% of 31,377 HTTPS websites of Alexa Top 1,000,000 have partial protections.
- **Analysis of existing defenses & new defense techniques.** We discuss pros and cons of potential defenses, and conclude that none of them provide full protection against BCP attacks. We provide guidelines for users and browser vendors to defeat such attacks completely, and propose defense techniques for website developers to mitigate cross-origin BCP attacks.
- **Systematic study & additional attack vectors.** We present a systematic study of BCP attacks against HTTPS. In addition to same-origin and cross-origin BCP attacks, we also identify a new attack vector: extension-assisted BCP attack vector.

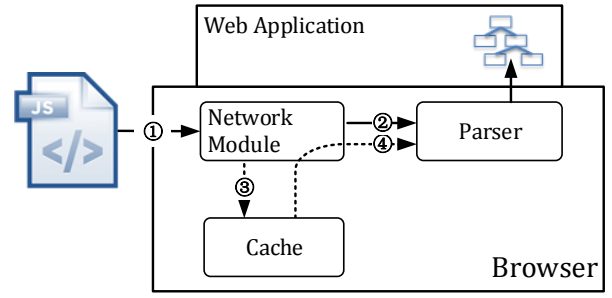


Figure 1: Illustration of loading resources in browsers via network (path ① and ② with solid lines) and via cache (path ③ and ④ with dotted lines).

tion to same-origin and cross-origin BCP attacks, we also identify a new attack vector: extension-assisted BCP attack vector.

2. OVERVIEW

2.1 Background: Browser Cache

The main purpose of browser cache is to reduce the loading time of web pages and resources. Current browsers employ memory cache and disk cache to store resources, e.g., HTML pages, JavaScript files, CSS files, PDFs and so on [7]. When the user requests such resources, the browser automatically loads the cached resources instead of sending requests to the server, as Figure 1 shows. The main types of browser cache are as follows.

Web Cache Shared across All Sites. Web cache is the default browser cache for all HTTP/HTTPS resources. By default, web cache is shared across all sites. Thus, once the browser caches a site’s resources over HTTP/HTTPS, if another site requests for the same resources, the browser loads the cached copies instead of issuing new requests. HTTP/1.1 provides *cache-control* and *expires* headers to specify the expiration time of caching resources, with the former having higher precedence [33]. During the specified lifetime, when cached resources are requested, the browser will not issue any GET requests for the resources until the expiry date or maximum age is reached. Thus once the attacker poisons the targeted site’s resources in web cache by setting the cache headers, the browser will directly load the cached resources from cache for all sites including these resources.

HTML5 AppCache Dedicated per Site. HTML5 introduces a new type of cache, i.e., HTML5 application cache (AppCache). With AppCache, an entire web application can be cached, including pages and resources, and they are accessible by the same origin for a long period of time even without Internet connection [12]. AppCache requires the web application to include a cache manifest that specifies which resources should be cached and which should not. When the web application is stored in AppCache, the browser will load the cached resources until the manifest file is changed or the AppCache is programmatically refreshed. Once the attacker crafts malicious copies for the targeted site or poisons the targeted site’s resources in AppCache, the victim’s browser will directly load the poisoned pages from application cache when the victim visits the targeted site even offline.

In this paper, we show that the caching policies implemented in various browsers allow the attacker to compromise any session and make persistent impact over time, when an SSL error is ignored by a user.

2.2 Threat Model

The adversary is a one-time MITM attacker against HTTPS, who intercepts HTTPS connections between Alice’s browser and the targeted site’s server only once. The MITM attacker can utilize a host of well-known MITM techniques (e.g., ARP poisoning and DNS pharming attacks) to re-route all the traffic of Alice to himself. To avoid either suspicion, or subsequently being blocked by additional security mechanisms, once the attacker completes the one-time MITM attack, he no longer intercepts the traffic from/to Alice. As a recent example, the Heartbleed vulnerability allows attackers to launch MITM attacks against HTTPS sessions, and security experts recommend a two-step fix to it [35]. However, as we show in this paper, a one-time MITM attack is sufficient for the adversary to persistently compromise the victim site’s future HTTPS sessions even after the fix.

We assume that the adversary mounts MITM attacks with forged certificates, e.g., self-signed or domain mismatch but authorized by trusted CAs. These are expected to raise SSL warnings to users. If attackers compromise CAs to forge certificates, e.g., security breaches of Comodo [2] and DigiNotar [4], no browser warnings will be raised to users when under MITM attacks.

We assume that when under a one-time MITM attack, Alice clicks through one SSL warning on a site over either HTTP or HTTPS. In reality, the majority of web users are inclined to click through SSL warnings on various scenarios. For example, a recent study shows that 70.2% of users choose to click through SSL warnings on various websites, e.g., google.com and facebook.com, on Chrome [15]. Dhamija et al. observe a 68% click-through rate, and Sunshine et al. even record 90%-95% click-through rates on various pages [26,54]. These studies focus on the SSL warning for top-level URL on the targeted site A over HTTPS. In addition to this warning, we also consider other three types of warnings, i.e., the SSL warning for A’s subresource shown on another site N, the warning for mixed content on A, and the SSL warning for one extension’s injected script shown on any site, which have not been well explored. In this work, we demonstrate the in-depth implication of a single click-through on an SSL warning. As we discuss below, browser cache poisoning attacks can compromise all future sessions with A, once Alice clicks through one warning of these four types.

2.3 Problem: Browser Cache Poisoning

In a traditional MITM attack against HTTPS, the network attacker has to intercept each session “on the wire” to compromise the targeted site over HTTPS. We demonstrate *browser cache poisoning (BCP)* attacks, which employ one-time MITM and have persistent effects. Based on scenarios of click-through warnings, we classify the browser cache poisoning attacks into three categories below.

Same-Origin. Suppose the targeted site over HTTPS is an online banking website. If Alice clicks through an SSL warning on the banking site, the attacker can impersonate as the site, and replace the page and the targeted subresources⁴ with his malicious ones. By setting long-lived cache headers, the attacker instructs Alice’s browser to store the malicious copies for a long time. The attacker can substitute the malicious resources for just the essential ones, e.g., `jquery.js`, which may be included in the bank’s login page. Alternatively, the attacker can utilize HTML5 AppCache to instruct Alice’s browser to store the malicious page, manifest, and subre-

sources in the dedicated storage for the banking site for one year or longer. Regardless of whether Alice is online or offline, when she revisits the banking site, her browser will directly load the whole page from AppCache without issuing any requests. Since the SSL warning occurs on the banking site, and this attack only affects the same site, we term such attack as *same-origin BCP* attack.

Cross-Origin. In a cross-origin attack, the attacker first injects the banking site’s HTTPS subresource into an HTTP response for another site, such as a news site. When Alice’s browser sends a request for the subresource, the attacker substitutes his malicious resource for the original one. Since he self-signs the banking site’s subresource with his certificate, Alice’s browser may raise a warning for the invalid certificate on the news site. If Alice clicks through the SSL warning, the damage is that the banking site over HTTPS is compromised by the news site over HTTP, even if the banking site’s HTTPS connection is unintercepted and no warnings are shown on the banking site in Alice’s browser. Even worse, if the poisoned subresource (e.g., `jquery.js`) is a common script library shared across numerous websites, all future sessions with these sites are compromised. Since the SSL warning shown on the news site is for the poisoned subresource, which is a cross-origin resource loaded in the banking site, we term such attack as *cross-origin BCP* attack.

Extension-Assisted. Targets for compromise can be further amplified by browser extensions. Many desktop browser extensions inject resources, e.g., scripts and CSS files, into every page. If Alice clicks through an SSL warning for one extension’s injected resource on any site over either HTTP or HTTPS, the attacker can poison it as explained above. The consequence is more devastating than previous two scenarios. Whenever Alice visits any site e.g., the banking site, the poisoned resource will be loaded into each visiting page. In addition, if the extension’s resource is over HTTP, the attacker can directly intercept the HTTP connection and replace it with malicious resources without causing any warnings in Alice’s browser. After that, if Alice clicks through the warning for mixed content (Safari and Opera do not have such warning) on the banking site, the poisoned HTTP resource will be loaded into the banking site, and the site over HTTPS is no longer secure.

As we have shown above, clicking through an SSL warning has persistent implications beyond the security of the present web session. Since this attack is based on poisoning the extension’s injected scripts, we term it as *extension-assisted BCP* attack.

We are not aware of any systematic study of BCP attacks on existing desktop and mobile browsers, though attacks via browser cache have been discussed in previous works [32,43,46,52,55,57] [18,20,47]. We introduce the new extension-assisted attack vector, and present a comprehensive study on BCP attacks.

SSL warnings’ presentations and browser caching policies vary a lot across browsers, and these variances have different impact on browser cache poisoning attacks. We will discuss the following research questions in next section.

1. What information is displayed in the warning, e.g., hijacked site’s URL and certificate?
2. What are the caching policies for resources over broken HTTPS on different browsers?
3. How many browsers, users and websites are susceptible to browser cache poisoning attacks?

3. BROWSER MEASUREMENT

We measure BCP attacks on five mainstream desktop browsers and 16 popular mobile browsers. We investigate the information displayed on warnings and caching policies across these browsers.

⁴The targeted subresources refer to external (not inline) resources that can alter the document content in the targeted site, e.g., JavaScript and CSS files, but not static resources, e.g., images.

These studies show that all evaluated browsers are susceptible to such attacks.

3.1 Experimental Methodology

We set up an Apache server as the attacker’s server, host the substituted resources for the targeted site’s resources on the server, and configure “Cache-Control:public, max-age=31536000” in the resources’ response headers to instruct browses to cache them for one year. We utilize mitmproxy [8] to intercept the traffic from/to the victim, replace the targeted site’s resources with malicious ones in the attacker’s server, and send the substituted responses to the victim’s browser. To forge certificates for target sites, we use OpenSSL to create custom self-signed SSL certificates for the targeted domain.

For the targeted site, because online banking websites contain users’ credential information, e.g., credit card number, these sites are often targeted by attackers. We choose Citibank’s website⁶ as the targeted site, and pick the site’s essential subresource⁷ for poisoning.

We mount BCP attacks on most popular browsers (e.g., Chrome, Firefox, Safari, Opera, IE, Maxthon, UC etc.) on various platforms (e.g., Mac OS X 10.9.3, Linux 12.04, Windows 7, Android 4.4.3, iOS 6, and Windows Phone 8). As Table 1 shows, these browsers cover over 99% desktop browser users, and more than 1,000,000,000 mobile browser users. We describe the details of our evaluation below.

3.2 The Inconsistency of SSL Warnings

The Secure Socket Layer (SSL) and its successor, Transport Layer Security (TLS) are the basis of the end-to-end security provided by HTTPS⁸. In a MITM attack against HTTPS, the attacker’s certificate either mismatches with the targeted site’s domain, or is self-signed that is not authorized by trusted CAs, thus the certificate is not trusted by the victim’s browser. Browsers usually prompt with SSL warnings to ask the victim whether to trust the certificate. This is the last defense of protecting HTTPS connections from MITM attacks. Once the user clicks through the warning, the browser will trust the attacker’s certificate, and the attacker can impersonate as the targeted site’s server.

However, as per our evaluation, browsers do not agree with each other on when and how to show such warnings. We gather SSL warnings and address bar warnings on various browsers and show them in our supplementary website [11]. We discuss the variances in information displayed for broken HTTPS resources below.

No SSL Warnings. One browser, i.e., CM browser (5.0.22), does not check the validity of certificates, and never shows SSL warnings for invalid certificates. As Figure 2 shows, CM browser always displays “Green Shield” in the address bar for all HTTPS connections, regardless of the invalid server-side certificate⁹. This browser has more than 10 million users on Google Play.

SSL warnings can be overlaid. The attacker can utilize click-

⁵For mobile default browsers, e.g., Safari and Android Default Browser, we use # of sold devices to represent # of users. For other browsers, we treat # of downloads in the official market as # of users.

⁶https://online.citibank.com/US/JSO/signon/LocaleUsernameSignon.do?locale=en_US

⁷<https://online.citibank.com/JFP/js/jquery/jquery-1.7.2.js>

⁸We use the more widely used term SSL to refer to both TLS and SSL in this paper.

⁹The vendor adds SSL warnings in the latest version, but CM is still vulnerable to BCP attacks when the user ignores the warning.

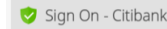


Figure 2: CM Browser always displays “Green Shield” for hijacked sites.



Figure 3: Chrome displays “Broken Lock” for sites with invalid certificates.

jacking [38] or tapjacking [50] to overlay and camouflage the SSL warning in an iframe, to further lure the victim to click through the warning. We have successfully tested this technique on Firefox 3.6, IE 8, IE 10 for Windows Phone and other old version browsers.

Incomplete SSL Warnings. As Table 1 shows, the majority of current browsers show SSL warnings for broken HTTPS connections with invalid certificates. For desktop browsers, whereas Firefox, Chrome and IE show in-page SSL warnings for hijacked sites, Safari and Opera display pop-up warnings. For mobile browsers, Firefox, Chrome, Opera, IE and UC display in-page warnings, but other 10 browsers (i.e., Android Default Browsers, Safari, Baidu, Maxthon, Next, Web Explorer Browser, Web Browser, Javelin, Dolphin and Boat), alert users with pop-up SSL warnings. All warnings have the same intended goal, i.e., to alert users that the server’s certificate is not trusted, but they have different presentations, e.g., various messages and appearances, as shown in Figure 4 and 5. We demonstrate the variances of the presented information below, and discuss their potential implications.

1) Default blocking or warning for hijacked subresources?

For cross-origin subresources with invalid certificates, Firefox, Safari, Chrome, Opera, IE for Windows Phone and UC directly block these resources from being loaded into the current page without showing any warnings. Other browsers, e.g., Android Default Browser, Baidu, Maxthon, Next, CM, Javelin, Dolphin, Boat, Web Explorer and Web Browser, prompt with SSL warnings shown in Figure 4 and 5 to caution users. For cross-origin BCP attacks, the attacker can inject the targeted site’s subresource into any site over HTTP. Since the warning appears on the HTTP site, e.g., news site, the user may be inclined to ignore it and continue browsing the site over HTTP. Once the subresource is hijacked by BCP attacks and cached in browsers, all the user’s future HTTPS sessions with the targeted site are compromised.

2) Missing the hijacked site’s URL in the warning. As Table 1, Figure 4 and 5 show, except Firefox, Chrome, Safari, Opera, IE (desktop and mobile version) and Android Default Browser, the other 10 mobile browsers (e.g., Dolphin with 50 million downloads) do not display the hijacked site’s URL in the warning. For Baidu, Maxthon, Next, Dolphin, and Boat browsers, after clicking “View certificate” and “View page info” buttons, the current page’s URL (not the hijacked subresource’s URL) will be shown. Since the hijacked subresource’s URL is missing in the warning, when under cross-origin or extension-assisted BCP attacks, the user may tend to notice that the warning is not for the current HTTP site and may be inclined to click through it.

3) No warnings for hijacked sites in the address bar. As Table 1 shows, when the user clicks through an SSL warning, all desktop browsers display warnings, e.g., “Broken Lock” in Chrome (as shown in Figure 3), in the address bar for sites with invalid certificates. For mobile browsers, though Baidu, Maxthon, Next, Javelin, Web Explorer and Web Browser have pop-up SSL warnings for hijacked sites, they do not display warnings in the address bar for these sites, e.g., “Green Shield” in CM browser as shown in Fig-

Table 1: SSL Warnings, Address Bar Warnings & Default Caching Policies in Mainstream Browsers

Mobile Browsers	I	II	III	IV	V	VI	VII	VIII	IX
Firefox (31.0)(Android)	50,000,000	✓	✓	✓	–	✓	✓	✓	✓
Chrome (36.0.1985.125)(Android & iOS)	500,000,000	✓	✓	✓	–	✓	✓	–	✓
Safari (7.0)(iOS)	800,000,000 [1]	✓	✓	✓	–	✓	✓	–	✓
Opera (22.0.1485.78487)(Android & iOS)	50,000,000	✓	✓	✓	–	✓	✓	–	✓
IE (10)(Windows Phone)	30,000,000 [9]	✓	–	✓	–	✓	✓	✓	✓
Android Default Browser(4.4.3) (Android)	1,000,000,000 [5]	✓	✓	–	–	✓	✓	✓	✓
Baidu (4.0.0.4)(Android)	10,000,000	✓	–	–	–	–	✓	✓	✓
Maxthon (4.2.6.2000)(Android)	5,000,000	✓	–	–	–	–	✓	✓	✓
Next (1.16)(Android)	5,000,000	✓	–	–	–	–	✓	✓	✓
CM (5.0.22)(Android)	10,000,000	–	–	–	–	–	–	✓	✓
Javelin (3.1.1)(Android)	100,000	✓	–	–	–	–	–	✓	✓
Web Explorer (2.0.6)(Android)	1,000,000	✓	–	–	–	–	–	✓	✓
Web Browser (1.2)(Android)	100,000	✓	–	–	–	–	–	✓	✓
Dolphin (11.1.6)(Android)	50,000,000	✓	✓	–	–	–	✓	✓	✓
Boat (7.7)(Android)	5,000,000	✓	✓	–	–	–	✓	✓	✓
UC (9.8.0)(Android)	50,000,000	✓	✓	✓	–	–	–	✓	✓
Desktop Browsers									
Firefox (31.0)(Linux, Windows & OS X)	15.54% [13]	✓	✓	✓	–	✓	✓	✓	✓
Chrome (36.0.1985.125)(Linux, Windows & OS X)	19.34%	✓	✓	✓	–	✓	✓	–	✓
Safari (7.0.4)(Windows & OS X)	5.28%	✓	✓	✓	–	✓	✓	–	✓
Opera (22.0.1471.70)(Linux, Windows & OS X)	1.05%	✓	✓	✓	–	✓	✓	–	✓
IE (10.0.9200.16540)(Windows)	58.38%	✓	✓	–	–	✓	✓	✓	✓

I : Market Share [13]/# of Users⁵ ✓ : Yes – : No

II : Show Pop-up/In-page SSL Warnings for Sites with Invalid Certificates

III: Show Address Bar Warnings for Sites with Invalid Certificates

IV: Block Cross-Origin Subresources with Invalid Certificates by Default

V : Show Address Bar Warnings for Cross-Origin Subresources with Invalid Certificates

VI: Display the Hijacked Site's URL in the SSL Warning

VII: Display the Invalid Certificate's Content in the SSL Warning

VIII: Cache Resources over Broken HTTPS in Web Cache

IX: Cache Resources over Broken HTTPS in AppCache

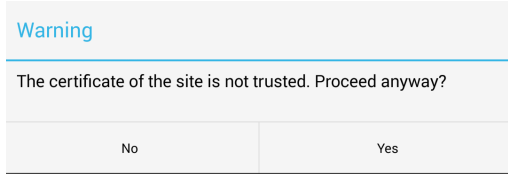


Figure 4: The SSL warning in Javelin displays incomplete information, e.g., missing the hijacked site's URL.

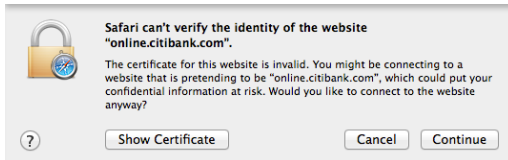


Figure 5: The SSL warning in Safari shows the hijacked site's URL and the certificate.

ure 2. For this case, once the targeted site is hijacked by the same-origin BCP attack on these browsers, these browsers will always load the substituted one without any warnings in the address bar.

4) **Missing the invalid certificate's content.** When HTTPS connections are intercepted, the certificate's content in the SSL warning can help users identify whether the certificate should be trusted or not. For example, in some enterprises, employees need to trust the company's certificate, even if it is invalid. Thus based on the certificate's content, the employee can determine to trust their company's certificate, but not hijacked sites' certificates. However, several mobile browsers, e.g., Javelin, Web Explorer, Web Browser and UC, do not display the certificate's content, which may confuse users to make blind decisions.

Such inconsistency among today's web browsers in warning users

of SSL errors has implications of clicking through the warnings. Especially, the improper warnings on certain mobile browsers, e.g., no warnings in the address bar and missing the hijacked site's URL, confuse users to be more susceptible to BCP attacks. Even worse, browsers' policies for caching resources when SSL errors are ignored by users' consent are heavily browser-specific. In Section 3.3, we discuss the incoherence of browser policies for broken HTTPS connections.

3.3 Incoherence of Browser Caching Policies

As we discuss in Section 3.2, SSL warnings are inconsistent among browsers and often do not provide enough information to caution users away from hijacked sites. If these browsers employ proper caching policies, they can limit the damage of BCP attacks to one session. However, our evaluation shows that caching policies for broken HTTPS connections are not consistent across browsers. For correct HTTPS connections, all browsers will respect the header's directives and cache the resources properly. For broken HTTPS connections after clicking through SSL warnings, different browsers deploy different caching policies. No specification defines this behavior properly.

Caching resources over broken HTTPS in web cache. As Table 1 shows, only Chrome, Safari and Opera do not cache resources over broken HTTPS in web cache, but all other browsers (e.g., UC with 50 million downloads) cache the resources. Since web cache is shared across all sites, if common JavaScript libraries are substituted by BCP attacks and cached in vulnerable browsers, all the sites that contain the same libraries are affected by such attacks. Meanwhile, once the JavaScript resources from extensions are replaced by BCP attacks and cached in web cache, all the pages opened in the future will be compromised.

Caching resources over broken HTTPS in HTML5 AppCache. Table 1 demonstrates that only Safari does not cache resources over Broken HTTPS in AppCache, but other desktop browsers and mobile browsers cache these resources. Thus once one site is under BCP attacks and substituted by a malicious one with a long-lived AppCache manifest, the victim’s browser will load the cached site for the specified duration.

No pop-up/in-page warnings for loading resources over broken HTTPS from browser cache. From our evaluation, we find that no browsers show pop-up/in-page warnings for loading resources over broken HTTPS from either web cache or AppCache. Once resources are cached, all these resources are trusted and there are no more integrity checks. Before these cached resources expire, browsers will load them for sites repeatedly without issuing new requests to fetch them from servers.

We have reported the incoherence of browser caching policies to the related browser vendors. Google acknowledges the importance of these findings and is deploying a fix in Chrome by not caching resources over broken HTTPS in HTML5 AppCache as we suggest in Section 5.

3.4 Summarizing Susceptibility of Browsers

Same-Origin. As Table 1 demonstrates, all browsers cache resources over broken HTTPS in either web cache or AppCache, except Safari on desktop platform. Thus for these browsers, once the victim clicks through one SSL warning on the targeted site, the browser will cache the substituted resources from the attacker, and the site is affected by same-origin BCP attacks.

Cross-Origin. Table 1 shows that only Chrome, Safari, and Opera do not utilize web cache to cache resources over broken HTTPS. Thus all the other browsers are affected by cross-origin BCP attacks. Because web cache is shared across different sites, web cache can be used by such attacks. As we discuss in Section 2.3, once the victim clicks through the warning for the targeted site’s subresource on any HTTP site, the targeted subresources can be replaced by the BCP attacker and cached in the victim’s browser for a long time. Thus the warning on any other site can affect the target site’s security.

Extension-Assisted. We observe that many extensions inject resources, e.g., JavaScript files, into every page, e.g., Free Smileys & Emoticons (1,820,058 users), Lightning Newtab (4,397,781 users) on Chrome, and WindowShopper (32,206 users) on Firefox. On Firefox, Chrome, Safari and Opera, we develop a tool to automatically download extensions and analyze their injection features. We show part of our results in Table 2. We mount extension-assisted BCP attacks on these extensions in four browsers and summary the results in Table 3. As Table 1 shows, only Firefox caches the injected JavaScript files over broken HTTPS, while Chrome, Safari and Opera do not. However, all these browsers allow extensions to inject HTTP scripts into sites over HTTPS. Safari and Opera do not display warnings for mixed content, and they will directly load the extension’s subresources over HTTP. Without the last line of defense in displaying the warning, the sites that contain mixed content can also be easily attacked. Once the extension’s subresources are poisoned by BCP attacks, all subsequently opened pages in Safari and Opera will be compromised. On the other side, Firefox and Chrome do not allow HTTPS pages to load subresources over HTTP by default, but users can override the default by clicking through a warning button for mixed content as Figure 6 shows. Once the victim clicks through the warning, these two browsers

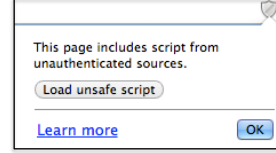
Table 3: Caching and Mixed Content Policies for Extensions’ Injected Resources on Chrome, Firefox, Safari & Opera

	I	II
Firefox (Linux, Windows & OS X)	✓	✓
Chrome (Linux, Windows & OS X)	–	✓
Safari (OS X & Windows)	–	✓
Opera (Linux, Windows & OS X)	–	✓

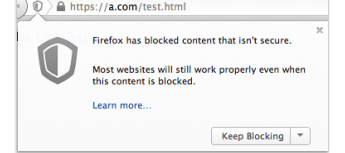
I : Cache Extensions’ Injected Resources over Broken HTTPS

II : Allow Extensions to Inject HTTP Resources into Sites over HTTPS

✓ : Yes – : No



(a) The warning for mixed content in Chrome.



(b) The warning for mixed content in Firefox.

Figure 6: The warnings for mixed content in Chrome 36 & Firefox 31. If users clicks through these warnings, Chrome & Firefox will load HTTP resources into HTTPS sites. Safari 7 and Opera 22¹¹ allow HTTPS sites to load mixed content by default, and do not have mixed content warnings.

will load the subresources over HTTP, which can be affected by our attacks.

In conclusion, all the evaluated browsers are susceptible to BCP attacks to some extent. Safari (desktop version) is only affected by extension-assisted BCP attacks, and Chrome and Opera (desktop and mobile version) are vulnerable to same-origin and extension-assisted BCP attacks. The other browsers, especially mobile browsers are susceptible to all the three series of attacks.

of Susceptible Users. Since all browsers we evaluate are vulnerable to BCP attacks to some extent, at least 99% desktop browser users and over 1 billion mobile browser users (as shown in Table 1) are susceptible to such attacks. We conduct our experiments on the browsers with relatively latest version as shown in Table 1, and we believe that the old version browsers are supposed to have the same problems and even severe ones.

Susceptible Websites in Alexa Top 100 & 1,000,000. Websites cannot completely thwart BCP attacks by themselves (discussed in Section 4), but by enabling proper settings, e.g., HSTS, they can partially mitigate such attacks. After our investigation on Alexa Top 100 websites, we find that 51 sites are over HTTPS, 22 sites set cache headers properly (e.g., no-cache), 2 sites (i.e., apple.com & bing.com) do not contain cross-origin resources, 6 sites set CSP headers¹² and 5 sites¹³ enable HSTS headers. Therefore, only 6 sites that set CSP, 5 sites that enable HSTS and 2 sites that do not contain cross-origin resources have better defenses to BCP attacks.

Furthermore, we send HTTPS requests to Alexa Top 1,000,000 sites to fetch their homepages, but only receive 31,377 responses. By analyzing the response headers of 31,377 HTTPS websites, we find that 510 (1.63%) sites enforce HSTS headers, 375 (1.20%) sites set cache-control headers, and only 45 (0.14%) sites enable CSP. The majority of HTTPS websites do not have any protection

¹⁰We collected the data in August, 2014.

¹¹Opera’s next version 23.0.1522.60 starts to support mixed content warnings.

¹²plus.google.com, facebook.com, twitter.com, mail.yandex.ru, pinterest.com and e.mail.ru set CSP headers.

¹³facebook.com, twitter.com, dropbox.com, paypal.com and alipay.com enable HSTS headers.

Table 2: Extensions/Add-ons that Inject Scripts into Every Page on Firefox, Chrome, Safari and Opera

Name	I	II	Name	I	II	Name	I	II
Free Smileys & Emoticons (C)	1,598,606	1	friGate-unlock sites (C)	265,191	2	Mini Clock (C)	8,243	9
Printer button (C)	7,164	7	Everplex Dark (C)	19,676	8	Video download helper (C)	467,430	9
3Dnator (C)	62,861	1	Pacman (C)	82,654	5	Iminent (C)	2,357,926	1
Emoji for Chrome (C)	768	3	PicShare (C)	94,693	3	Album Downloader (C)	64,415	10
EXIF Viewer (C)	39,674	1	Imageshack-Clickberry (C)	23,020	1	Dailymotion downloader (C)	15,957	3
Lightning Newtab (C)	4,397,781	1	Search Switch (C)	307,889	9	Search All (C)	305,701	9
ShopperPro (C)	305,386	11	Slick Savings (C)	470,422	5	Shopping Helper (C)	575,752	4
uTorrent for Chrome (C)	89,564	4	San Antonio Spurs (C)	3,111	6	Boston Red Sox (C)	2,720	6
St. Louis Cardinals (C)	1,773	6	Chicago Blackhawks (C)	1,613	6	San Francisco 49ers (C)	1,658	6
New York Yankees (C)	1,554	6	Los Angeles Dodgers (C)	1,344	6	Los Angeles Angels (C)	314	6
Chicago Cubs (C)	738	6	St. Louis Blues (C)	241	6	Detroit Tigers (C)	1,095	6
MLB.com (C)	1,141	6	San Francisco Giants (C)	1,038	6	Ohio State (C)	938	6
Atlanta Braves (C)	859	6	Save My Ass !!! (F)	83	2	Moujazz Summary (F)	115	2
Curiyo (F)	1,586	5	paintitgreyscale (F)	11	1	WindowShopper (F)	32,206	7
X-notifier (S)	-	4	PinChoose (S)	-	5	Rundavoo (S)	-	2
Emoticons For FB (F)	5,688	2	FreeStyler (O)	20,238	1	Tawea (O)	1,432	4

I : # of Users¹⁰ II : # of Injected Scripts – : No Statistics C : Chrome F : Firefox S : Safari O : Opera

Table 4: Various Techniques for Mitigating Browser Cache Poisoning Attacks

	I	II	III
HSTS [36]/HPKP [28]	✓	–	–
Channel IDs [16]/SISCA [45]	–	–	–
DANE [37]/CAA [34]	–	–	–
CSP [3]	–	–	–
Web Cryptography [25]/Subresource Integrity [19]	–	✓	–
Private Browsing Mode [14]	–	–	–
Randomization of Resources' URLs [42]	–	✓	–
Segregating Browser Cache [41]	–	✓	✓

I : Same-Origin Browser Cache Poisoning Attacks

II : Cross-Origin Browser Cache Poisoning Attacks

III: Extension-Assisted Browser Cache Poisoning Attacks

✓ : Mitigate – : Not Mitigate

against BCP attacks.

4. INSUFFICIENCY OF EXISTING SOLUTIONS

One straightforward defense for browsers is not to cache resources over broken HTTPS. Since the hijacked resources over HTTPS cannot be cached in browsers, this solution can prevent browser cache poisoning attacks over HTTPS. As Table 1 shows, Chrome, Opera, and Safari have already implemented this caching policy for web cache, but only Safari deploys it for HTML5 AppCache. However, the majority of mobile browsers, e.g., Android Default Browser, Firefox for Android and Maxthon, do not apply this policy and are susceptible to both kinds of poisoned caches.

Various existing defenses against HTTPS attacks or attacks via browser cache can help defend against BCP attacks. However, they are not sufficient. As Table 4 summarizes, CSP [3], Channel IDs [16], SISCA [45], DANE [37], CAA [34], and private browsing mode [14] cannot thwart any series of BCP attacks; HSTS [36] and HPKP [28] can mitigate same-origin BCP attacks; Web Cryptography API [25], Subresource Integrity [19] and randomization of resources' URLs [42] prevent cross-origin BCP attacks; segregating browser cache [41] protects users from cross-origin and extension-assisted BCP attacks. Therefore, none of these techniques provide comprehensive protection against BCP attacks. We describe details below.

4.1 Defenses against MITM Attacks

Strict Transport Security (HSTS) & Public Key Pinning (HPKP). HSTS [36] is the successor of ForceHTTPS [40], which is proposed to mitigate SSL stripping attacks [49]. It provides an HTTP re-

sponse header for a website to force browsers to make SSL connections mandatory for all subresources on this site. Once HSTS is set in the HTTP header, none of HSTS-compliant browsers give users the option to ignore SSL errors. However, for HSTS, browsers must first connect to the legitimate websites securely to fetch the authorized certificates before connecting to untrusted networks [39]. Thus if the BCP attack occurs before the victim connects to the legitimate site, the attacker can still poison the targeted site's resources. After testing four sites that enable HSTS headers on Firefox, i.e., facebook.com, github.com, paypal.com and alipay.com, we find that the HSTS headers can be stripped by the attacker if it is the user's first visit, and after that the sites are not protected by HSTS.

Public Key Pinning (HPKP) [28] allows websites to specify their own public keys with a HTTP header, and instructs browsers not to accept any certificates with unknown public keys. Without connecting to the legitimate websites securely for the first time, some browsers, e.g., Chrome and Firefox, pre-load the public keys for well-known websites, e.g., google.com, to deploy HPKP or HSTS [6, 10]. While current browsers only pre-load public keys of selected sites, it is impractical for them to pre-load the public keys of all sites over HTTPS. Both HSTS and HPKP instruct browsers to cease connections with servers over broken HTTPS to protect these sites from MITM attacks. However, if the targeted site contains cross-origin subresources that are not protected by HSTS/HPKP, these resources can be poisoned by cross-origin BCP attacks over broken HTTPS. We conduct experiments on two sites with HSTS headers (i.e., github.com and twitter.com), which both contain `https://www.google-analytics.com/analytics.js` without HSTS headers. We find that after poisoning analytics.js, when visiting these two sites over correct HTTPS, the browser will load the hijacked script into these two sites without any warnings. For extension-assisted BCP attacks, HSTS/HPKP cannot prevent browsers from loading the hijacked extensions' scripts over HTTP/HTTPS into the targeted site. Meanwhile, currently the majority of mobile browsers, e.g., IE 10 (Windows Phone), Android Default Browser, Baidu, Maxthon, Next, CM, Javelin, Web Explorer, Web Browser, Dolphin, Boat, and UC, do not support HSTS/HPKP.

Channel IDs & SISCA. Channel IDs [16] is a TLS extension, which was originally proposed as Origin-Bound Certificates (OBCs) [27]. Channel IDs enables browsers to generate self-signed certificate to conduct TLS client-side authentication, and further prevent MITM attackers to impersonate as victims' browsers. Server Invariance with Strong Client Authentication (SISCA) [45] combines Channel IDs-based client authentication and server invariance to protect

against MITM attackers who impersonate the user to the server. However, the attackers discussed in this work impersonate the server to the user, and therefore Channel IDs/SISCA do not prevent BCP attacks.

In particular, BCP attacks can compromise SISCA’s guarantees. To prevent resource caching poisoning, SISCA sets the *ETags* header to instruct browsers to check the integrity of the cached resource, and sets the *If-Non-Match* header to verify that the local version matches the lastest version on the server. Nevertheless, these settings are in response headers, which can be easily replaced by the BCP attacker when poisoning the targeted resources with setting long-lived cache headers. The attacker can also poison the cross-origin subresources or the extension’s injected resources in the targeted site. Therefore, when the user visits the targeted site, the browser will load these malicious cached resources rather than the original ones. The poisoned resources, e.g., JavaScript, have unrestricted access over the credentials belonging to the site on behalf of the user. Thus SISCA cannot mitigate BCP attacks.

DANE & CAA. The Certification Authority Authorization (CAA) DNS Resource Record [34] allows a DNS domain name holder to specify Certification Authorities (CAs) authorized to issue certificates for that domain. DNS-based Authentication of Named Entities (DANE) [37] enables the administrators of domain names to sign SSL certificates for websites on their domains. Nevertheless, these approaches are based on DNSSEC, which are not widely deployed on the Internet. Furthermore, these solutions do not impel browsers not to cache resources over broken HTTPS, thus they cannot mitigate BCP attacks.

4.2 Content Restriction & Document Integrity

Content Security Policy (CSP). CSP [3] provides HTTP headers for a website to declare approved resources (e.g., JavaScript, CSS, frames, etc.), which are whitelisted to be loaded on the page in browsers. Other resources that violate the policy will be blocked and reported to the site. CSP helps detect and mitigate cross site scripting (XSS) and some subresource-injection attacks. However, CSP is a parser-level defense, and it does not check the integrity of a resource. With preserving the same URLs, cross-origin BCP attacks replace approved subresources with malicious ones, thus CSP cannot detect the differences and mitigate such attacks. Furthermore, same-origin BCP attacks can hijack the whole site and substitute the forgery site without CSP headers for the original one. In addition, browser extensions are exempt to CSP, and can inject scripts into websites regardless of the origins of the scripts [53]. Thus CSP does not interfere with extension-assisted BCP attacks. Overall, CSP does not provide any prevention against BCP attacks.

Web Cryptography API & Subresource Integrity. Web Cryptography API [25] provides a JavaScript API for performing basic cryptographic operations in web applications, e.g., encryption, decryption, hashing, and signature generation. Subresource Integrity [19] introduces a mechanism for browsers to verify that subresources in web applications have been delivered without unexpected manipulation. Subresource Integrity extends several HTML elements with a *integrity* attribute that contains a cryptographic hash of the representation of the resource below.

```
<script src='https://www.google-analytics.com/analytics.js'
  integrity='ni:///sha-256;ptdSz0i-j-P9_TJ-CmNY-YXjDzeQL5UbgNQJlKoCAA=?ct=application/javascript'>
</script>
```

Both Web Cryptography API and Subresource Integrity provide the functionality for browsers to check data integrity for subresources.

Thus browsers can realize that the poisoned subresources are not the same ones in the original site, which mitigates cross-origin BCP attacks. However, for same-origin BCP attacks, the attacker can replace the subresource, re-compute the hash value and set the new one in the targeted site. For extension-assisted BCP attacks, the injected scripts from extensions are beyond the control of these two techniques. Thus Web Cryptography API and Subresource Integrity cannot defeat these two attacks. These two techniques are still in W3C working drafts, which are not supported by any browsers at this moment.

4.3 Defenses via Browser Cache

Private Browsing Mode. Private browsing modes¹⁴ prevent browsers from permanently storing any cookies, histories, caches or other site related states. However, during private browsing mode, browsers still cache resources of different websites [14]. Browsers only clear the cached resources after closing windows by users. Thus when browsers are in private browsing mode before closing, they are still susceptible to BCP attacks. For Chrome and Opera, they disable extensions in private browsing mode by default, thus they partially prevent extension-assisted BCP attacks. Meanwhile, comparing to desktop browsers, many mobile browsers, e.g., CM, UC and Dolphin, do not support private browsing mode.

Randomization of Resources’ URLs. Randomization of resources’ URLs instructs client-side browsers not to cache these resources by adding a unique random string in each resource’s URL, e.g., `www.google-analytics.com/analytics.js?19991`. Thus when a user visits the targeted site, the site includes a different URL for the same resource, and the user’s browser always fetches the latest one from the server instead of loading from cache. Jakobsson et al. neutralize attacks via browser cache by means of URL personalization with this idea [42]. As users cannot predict all the URLs, the targeted site will provide at least one static URL for a starting page. Thus users can visit the site by typing the URL in the address bar or from search results on search engine. Since the attacker cannot predict URLs of the targeted site’s subresources, cross-origin BCP attacks cannot work. Nevertheless, in same-origin BCP attacks, the attacker can substitute a malicious page for the targeted site’s starting page to compromise the future sessions. Meanwhile, the extension’s hijacked resources are not obfuscated and still cached in the victim’s browser. Therefore, this technique does not defeat the same-origin and extension-assisted attack vectors, but protects against cross-origin attacks.

Segregating Browser Cache. Jackson et al. proposed to deploy Same-Origin Policy on browser cache to prevent websites from loading cached resources from other sites [41]. This approach prevents hijacked resources from being shared across different sites, and every site can only load its own cached resources. Therefore, this technique thwarts cross-origin and extension-assisted BCP attacks, but not same-origin BCP attacks. However, Jia et al. demonstrate that this defense introduces significant performance overhead [43]. That may be one reason why this solution has not been adopted by current browsers in the default setting.

5. OUR DEFENSE TECHNIQUES

As we discussed in Section 4, no existing defenses provide full protection against browser cache poisoning attacks. In this section, we first discuss guidelines for users and browser vendors to defeat

¹⁴Private browsing modes represent Private Browsing in Safari and Firefox, Incognito Mode in Chrome, Private Window in Opera, and inPrivate Browsing in IE

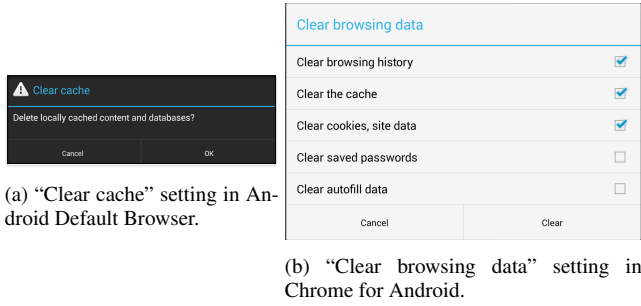


Figure 7: "Clear cache" setting in Android Default Browsers does not specify web cache and HTML5 AppCache. "Clear browsing data" setting in Chrome provides various options for users to clear browsing data.

BCP attacks. However, user faults and browser implementation errors are the main reasons for browser cache poisoning attacks. We then propose defense techniques for web developers to mitigate cross-origin BCP attacks with minor performance overhead.

Guidelines for Users. Users should not click through SSL warnings on any site in normal browsing mode. As a precaution, they should also clear browser cache, i.e., web cache and HTML5 AppCache, before visiting a site requesting credentials, especially after an SSL warning is clicked.

After investigating the settings of 21 browsers, we find that Javelin, Web Explorer and Web Browser do not provide the option for users to clear cache. Safari (mobile and desktop version), IE (Windows Phone version), Android Default Browser and Maxthon have the "Clear cache" button as shown in Figure 7a, but the setting does not specify web cache and AppCache. The other browsers, e.g., Chrome and Firefox, support various options for users to clear browsing data as shown in Figure 7b. However, clearing cache takes several steps. For example, on Chrome (Android version), users need to click "Setting", "Privacy", and "Clear browsing data" to trigger the clearing.

However, even if users follow the setting to clear cache, Baidu, Next, Javelin, Web Explorer, Web Browser and CM do not clear AppCache. Once an attacker poisons the targeted resources in AppCache, these six browsers will cache the malicious resources until the user uninstalls them. Therefore, never clicking through any SSL warnings is the only proper way for users protect themselves from BCP attacks.

Guidelines for Browser Vendors. From the perspective of a browser vendor, to completely defeat BCP attacks, there are two requirements that suffice: (1) No caching for resources over broken HTTPS in either web cache or AppCache; (2) Default blocking sites over HTTPS from loading HTTP resources.

The first requirement protects users from same-origin, cross-origin, and extension-assisted BCP attacks over HTTPS, and the second one prevents the extension-assisted attack vector over HTTP. As Table 1 depicts, only Safari (desktop version) meets the first requirement, but other browsers especially mobile browsers do not provide such protection. For Chrome, after receiving our report, Google has confirmed the vulnerability in AppCache and is deploying a fix to meet the first requirement. For the second policy, Chrome and Firefox (desktop version) block mixed content by default with warnings as shown in Figure 6, but other browsers do not have such policy. By implementing these two policies, browsers can protect users from BCP attacks without the server-side modification and the assistance from users.

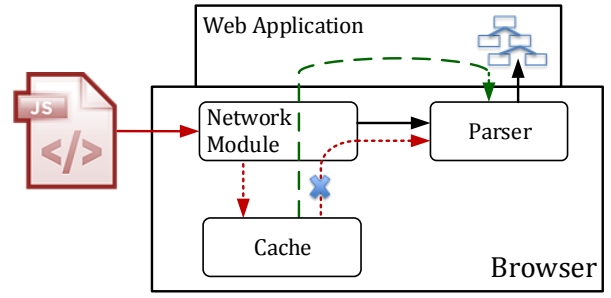


Figure 8: Illustration of the defense we propose, which ensures only fresh and unpoisoned subresources can be loaded into the targeted site's page. As red dotted lines depict, the poisoned JavaScript subresources can be directly loaded from browser cache by default. In our approach (as green dashed lines show), the targeted site checks the integrity of all cached subresources before loading them into the page from cache.

Defense Techniques for Website Developers

As we discussed in Section 3, the inconsistency of SSL warnings and the incoherence of caching policies increase the vulnerability to BCP attacks. Websites cannot impel all browsers to implement proper protections or force all users to use the upgraded browser. Thus for websites with users' credential information, e.g., online banking sites, they have the responsibilities to protect users from BCP attacks even without browsers and users' additional cooperation.

To defeat extension-assisted BCP attacks without the support from browsers and users is difficult. However, most desktop browser extensions do not inject scripts into every page, and all mobile browsers do not support extensions, which alleviates such threats. Users may be more inclined to click through warnings on the sites with which users do not exchange any sensitive information, e.g., news and blog sites, than on the sensitive sites, e.g., online banking sites. The cross-origin BCP attack makes it a powerful vector for exploits. Thus the cross-origin attack vector is the most deluding vector and can affect most users comparing to the other two vectors.

As discussed in Section 4, Web Cryptography API, Subresource Integrity, segregating browser cache and randomization of all resources' URLs can mitigate cross-origin BCP attacks. However, the first three defenses currently are not deployed on any browsers, and require browser vendors to modify source code and add these new features. The last technique impels browsers not to cache any resources at the expense of increased performance overhead. We propose a balanced approach to mitigate cross-origin BCP attacks with minor performance overhead (<5%), which works on all commodity browsers without browser's modification.

Approach Overview. Our main goal is to prevent the targeted site from loading the poisoned JavaScript subresources in the user's browser cache. As Figure 8 illustrates, in our approach, the targeted site checks the integrity of all cached JavaScript subresources before loading them into the page. Therefore, only fresh and unpoisoned subresources can be loaded into the targeted site's page.

For any subresource B included in the targeted site's page, we use inline scripts to follow the procedure in Algorithm 1. By sending two XMLHttpRequests for B, the inline scripts check the caching status of B by timing techniques. If B is not cached, the scripts append B with the original URL into the page; otherwise, the scripts check the integrity of B. If B is through the check, which indicates B is not poisoned, the scripts append B with the original URL into the page; otherwise, B is poisoned, thus the scripts append B with B's URL and a random string into the page, which triggers

Data: Suspicious subresource
Result: Load sanitized subresource
let B be a subresource.
Given B , **check** the caching status of B by timing techniques.
if B is not cached **then**
 Append B with the original URL into the page;
else
 compute $SHA_{256}(B)$;
 check the Integrity of B ;
 if B passes the check **then**
 Append B with the original URL into the page;
 else
 /* B is poisoned; */
 Append B with B 's URL and a random string into the page;
 Fetch the latest version of B from the server;
 end
end

Algorithm 1: Check the Caching and Integrity Status of a Subresource

the browser to fetch the latest version of B from the server. Since the browser never load the poisoned subresources into the targeted site's page, this approach mitigates cross-origin BCP attacks. We describe the implementation details below.

Implementation Details. Suppose the targeted web application is a website A with the domain $a.com$. We configure TLS to enable that A is over HTTPS. For infrequently changed resources, e.g., common libraries, we set long-lived cache headers for them. For other resources, e.g., dynamic JavaScript files, we set no-cache headers for them, and add random strings in the URLs in case that browsers do not support cache headers.

1) For the resources with random strings in the URLs, we directly add them as external scripts in the page. For the infrequently changed JavaScript files, we use inline scripts in every page to send XMLHttpRequests to the server to fetch these resources before inserting them. When fetching the JavaScript subresources, we use inline scripts to check the caching status of these scripts. We set the start time in the *onloadstart* event handler, and set the end time in the *onreadystatechange* event handler. We measure two rounds of the request load time of each subresource. If the time difference is larger than the threshold, e.g., 100 ms, it indicates a cache miss for the subresource. If the request load time is approximately same for two rounds, the subresource is cached in the user's browser.

Below is the piece of code to measure the load time of XMLHttpRequests.

```
var startTime, endTime, loadTime;
var xmlhttp = new XMLHttpRequest();
xmlhttp.onloadstart = function(){
    startTime = (new Date()).getTime();
}
xmlhttp.onreadystatechange = function(){
    endTime = (new Date()).getTime();
    loadTime = endTime - startTime;
    .....
}
```

2) Based on the caching status of each JavaScript subresource, we have different logic to handle it. The client-side scripts fetch the file C containing the latest SHA256 values for infrequently changed subresources from the server via a URL containing a random string. If the subresource B is not cached, which indicates that it is not poisoned by the attacker, the inline scripts directly add B in the page as an external script without the integrity check to reduce performance overhead. Otherwise the scripts calculate the SHA256

hashing value for B . If the hashing value equals to the value for B in C , which means that B is not poisoned, the inline scripts append B as an external script in the page. Otherwise, B is poisoned, and the inline scripts will add B with B 's URL and a random string in the page, which automatically fetches B 's latest version from A 's server. Below is the piece of code to handle different cache and integrity status for one subresource.

```
var xmlhttp, loadTime, loadTimeOld, threshold, realHash,
    link, url;
var xmlhttp = new XMLHttpRequest();
var rand = Math.floor((Math.random() * 1000000000) + 1);
var head = document.getElementsByTagName("head")[0];
var script = document.createElement("script");
script.type = "text/javascript";
if (Math.abs(loadTime - loadTimeOld) < threshold){
    //cached
    var hash = CryptoJS.SHA256(xmlhttp.responseText);
    if (realHash == hash){
        url = link;
    }
    else{
        url = link + "?" + rand;
    }
}
else{
    //not cached
    url = link;
}
script.src = url;
head.appendChild(script);
```

3) By default the client-side scripts in A can only issue XMLHttpRequests to fetch A 's resources but not the resources from other domains. Although the "Access-Control-Allow-Origin" header loose the restriction to allow other domains to access the resource, few resources set the header as "*" (allow all sites to access the resources) or explicitly specify $a.com$ as a privileged domain.

To solve this problem, by setting a *reverse proxy*, we enable the web server to provide cross-origin resources with URLs under $a.com$. Take apache as an example, we enable the proxy module and set "ProxyPass /service/ https://www.google-analytics.com/" in the configuration file. After the setting, in any page of A , `<script src="/service/analytics.js"></script>` equals `<script src="https://www.google-analytics.com/analytics.js"></script>`. Therefore, `www.google-analytics.com/analytics.js` is transparently hosted on $a.com$. We configure the reverse proxy and convert the URLs of all cross-origin resources in A , e.g., third-party analytics scripts, common libraries, and advertisement resources, to the URLs under $a.com$. Thus the client-side scripts in A can fetch cross-origin resources under $a.com$ with XMLHttpRequests. To avoid introducing security loopholes, the reverse proxy only processes such resource requests that 1) come from A , and 2) fetch a selected set of URLs maintained by A 's developers.

As we describe above, in the targeted site A , all JavaScript subresources can be classified into four categories: never cached resources with URLs containing a random string, not cached ones with normal URLs, the resources that are cached and pass the integrity check with normal URLs, and cached but poisoned ones with random URLs. Since the subresources with random URLs cannot be predicted by the browser cache poisoning attacker, and the ones with normal URLs are not poisoned, this approach mitigates cross-origin BCP attacks.

Performance Evaluation. To understand the performance impact of our proposed technique, we applied it to 10 popular web applications within two days. Since attackers usually compromise login pages to steal users' credentials, we use the login page of each web application to measure the performance overhead. We fetch the login page of these 10 websites and host them on our server. We

Table 5: Page Load Time for the Original Login Page & the Modified One (in milliseconds) with Browser Cache

Website	Time (Original)	Time (Modified)	Overhead
google.com	779	810.5	4.04%
facebook.com	467	487.8	4.45%
youtube.com	2134	2235.8	4.77%
yahoo.com	1523	1587	4.20%
twitter.com	331	346.9	4.80%
linkedin.com	1340	1387	3.51%
dropbox.com	1225	1265.9	3.34%
paypal.com	548.5	574.1	4.67%
github.com	723.6	752.7	4.02%
workpress.com	1652	1712.5	3.66%

retrofit these websites to adopt our solution, measure the page load time of the original page and the modified one (averaged on 10 runs).

Table 5 summarizes the results of page load time for the original login page and that for the modified one. We can see our solution introduces the negligible performance overhead (<5%) to these websites. Different from randomization of all resources’ URLs, we only randomize the poisoned resources’ URLs and the browser can still load unpoisoned resources from cache. Thus our approach causes minor performance overhead.

6. RELATED WORK

There has been extensive research on attacks on browser cache and HTTPS/SSL connections, as well as the corresponding defenses. The browser cache poisoning attacks discussed in this work exposes HTTPS web sessions to far more persistent threat, and thus require more sophisticated countermeasures beyond existing solutions.

Attacks via Browser Cache. Felten et al. and Bortz et al. deploy timing attacks on browser cache to sniff users’ browsing histories and steal private information [18, 32]. Wondracek et al. de-anonymize social network users by analyzing users’ visited URLs [57]. More recently, Jia et al. show that timing attacks on browser cache can also be used to infer victim users’ geolocations [43]. On the other hand, researchers have also examined attacks by poisoning web cache, HTML5 AppCache, and other storage [44, 46–48, 52, 56] [20, 55].

Although some of the attack vectors discussed in this paper have been briefly experimented in previous studies, in this paper, we provide the first in-depth evaluation of the susceptibility of desktop and mobile browsers to all three BCP attack vectors, as well as a comprehensive analysis on whether existing solutions can mitigate such attacks. Our evaluation results raise serious concern on the security of HTTPS web sessions with all of today’s popular browsers. We further propose novel defense techniques for websites to protect their sessions immediately before browsers might adopt any BCP prevention mechanisms in future.

Defenses against Browser Cache Attacks. To prevent privacy leakage via browser cache, Jackson et al. propose a refined same-origin policy to segregate browser cookie and cache to protect browser states [41]. Jakobsson et al. neutralize browser sniffing by performing URL personalization on the fly at the server side [42]. Jia et al. [43] advocate not to cache location-sensitive resources to prevent leaking users’ geolocations. However, as we discuss in Section 4, defenses on browser cache alone cannot prevent browser cache poisoning attacks.

Clicking through of SSL Warnings. When an SSL warning is shown for a web page, the user is supposed to close the page to

protect him/her from MITM attacks. However, 33.0% and 70.2% of users choose to click through SSL warnings on various websites in Mozilla Firefox (beta channel) and Google Chrome (stable channel) respectively, according to Akhawe et al [15]’s investigation. Dhamija et al. observe a 68% click-through rate, and Sunshine et al. even record 90%-95% click-through rates depending on the type of page [26, 54]. In addition, Sunshine et al. find that many respondents do not understand SSL warnings, so they simply ignore the warnings [54]. These studies demonstrate that users easily click through SSL warnings. In this paper, we present BCP attacks after users’ clicking through SSL warnings and show that ignoring warnings can bring disastrous damage to the security and privacy of their web sessions.

Attacks against HTTPS. Prior research has unravelled numerous attacks to compromise HTTPS [17, 21–23, 45, 49, 51]. For example, Karapanos et al. present Man-In-The-Middle-Script-In-The-Browser (MITM-SITB) attacks to bypass enhanced Channel IDs-based defenses [45]. Fahl et al. mount MITM attacks on mobile applications to analyze SSL security in Android [29] and iOS [30]. In this work, instead of examining ways to directly thwart HTTPS security, we focus on the implications of one-time compromise of an HTTPS session. We show that the effect is persistent compromise of the victim’s future sessions, far beyond the boundary of the particular web session or even the website.

Defenses against HTTPS attacks. On the defense side, numerous researchers propose various solutions to protect HTTPS connections from attacks [3, 16, 17, 19, 24, 25, 27, 28, 34, 36, 37, 39, 40, 45]. However, as we elaborate in Section 4, none of these existing solutions prevent our browser cache poisoning attacks completely.

7. CONCLUSION

In this paper, we present a systematic study of browser cache poisoning attacks against HTTPS, which persistently compromise the victim’s future web sessions with the targeted site by poisoning the victim’s browser cache. Through experiments on five mainstream desktop browsers and 16 popular mobile browsers, we find the inconsistency of SSL warnings and incoherence of browser caching policies. In particular, the majority of mobile browsers, e.g., Android Default Browser, do not deploy SSL warnings properly, and always cache resources over broken HTTPS. In our evaluation, we demonstrate that all 21 browsers that cover over 99% desktop browser users and 1,000,000,000 mobile users, are susceptible to BCP attacks. Meanwhile, only five sites of Alexa Top 100 and 1.63% of 31,377 HTTPS websites have partial protections. Furthermore, we discuss pros and cons of potential defenses, and provide guidelines for users and browser vendors to defeat BCP attacks. We also propose defense techniques for website developers to mitigate these attacks.

8. REFERENCES

- [1] Apple has sold more than 800 million ios devices, 130 million new ios users in the last year.
<http://goo.gl/ZqBPvk>.
- [2] Comodo report fraudulently issued certificates.
<http://goo.gl/IgwXcZ>.
- [3] Content security policy. <http://goo.gl/Ivoq31>.
- [4] Diginotar reports security incident.
<http://goo.gl/d18E5e>.
- [5] Google: 1 billion people using android devices.
<http://goo.gl/cBL1B9>.
- [6] Http strict transport security.
<http://www.chromium.org/sts>.

- [7] Leverage browser caching. <http://goo.gl/OE49jD>.
- [8] mitmproxy: a man-in-the-middle proxy. <http://mitmproxy.org/>.
- [9] Nokia: 50 million windows phone sales possible for 2014 (nok). <http://goo.gl/150hdP>.
- [10] Preloading hsts. <http://goo.gl/4jFTE7>.
- [11] Ssl warnings and address bar warnings in various desktop and mobile browsers. <http://goo.gl/RfpJf2>.
- [12] Using the application cache. <http://goo.gl/6vrLt9>.
- [13] Windows 8.x, internet explorer both flatline in june. <http://goo.gl/5I1Ysh>.
- [14] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, 2010.
- [15] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symposium*, 2013.
- [16] D. Balfanz and R. Hamilton. Transport layer security (tls) channel ids. <http://goo.gl/uasmHw>.
- [17] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy*, 2014.
- [18] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *World Wide Web*, 2007.
- [19] F. Braun, D. Akhawe, J. Weinberger, and M. West. Subresource integrity. *W3C Working Draft*, 2014.
- [20] E. Bursztein, B. Gourdin, G. Rydstedt, and D. Boneh. Bad memories. *BlackHat*, 2010.
- [21] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1), 2009.
- [22] S. Checkoway, M. Fredrikson, R. Niederhagen, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of dual ec in tls implementations. In *USENIX Security Symposium*, 2014.
- [23] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang. Pretty-bad-proxy: An overlooked adversary in browsers' https deployments. In *Security and Privacy*, 2009.
- [24] I. Dacosta, M. Ahamad, and P. Traynor. Trust no one else: Detecting mitm attacks against ssl/tls without third-parties. In *Computer Security—ESORICS*. 2012.
- [25] D. Dahl and R. Slevi. Web cryptography api. *W3C Working Draft*, 2013.
- [26] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Conference on Human Factors in Computing Systems*, 2006.
- [27] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security Symposium*, 2012.
- [28] C. Evans and C. Palmer. Public key pinning extension for http. <http://tools.ietf.org/html/draft-ietf-websec-key-pinning-19>, 2011.
- [29] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Computer and Communications Security*, 2012.
- [30] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In *Computer and Communications Security*, 2013.
- [31] A. P. Felt, R. W. Reeder, H. Almuhiemedi, and S. Consolvo. Experimenting at scale with google chrome's ssl warning. In *Conference on Human factors in Computing Systems*, 2014.
- [32] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Computer and Communications Security*, 2000.
- [33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol-http/1.1, 1999. <http://www.rfc.net/rfc2616.html>.
- [34] P. Hallam-Baker and R. Stradling. Dns certification authority authorization (caa) resource record. <http://tools.ietf.org/html/rfc6844>.
- [35] N. Henderson. How to fix heartbleed in two steps, and other security threats to watch out for. <http://goo.gl/jzAALz>.
- [36] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). <http://goo.gl/1b7c6D>.
- [37] P. Hoffman and J. Schlyter. The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa. Technical report, 2012.
- [38] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, 2012.
- [39] L.-S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged ssl certificates in the wild. In *Security and Privacy*, 2014.
- [40] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *World Wide Web*, 2008.
- [41] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *World Wide Web*, 2006.
- [42] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In *World Wide Web*, 2006.
- [43] Y. Jia, X. Dong, Z. Liang, and P. Saxena. I know where you've been: Geo-inference attacks via the browser cache. *Web 2.0 Security and Privacy*, 2014.
- [44] M. Johns, S. Lekies, and B. Stock. Eradicating dns rebinding with the extended same-origin policy. In *USENIX Security Symposium*, 2013.
- [45] N. Karapanos and S. Capkun. On the effective prevention of tls man-in-the-middle attacks in web applications. In *USENIX Security Symposium*, 2014.
- [46] A. Klein. Web cache poisoning attacks. In *Encyclopedia of Cryptography and Security*. 2011.
- [47] L. Kuppan. Attacking with html5. *BlackHat*, 2010.
- [48] S. Lekies and M. Johns. Lightweight integrity protection for web storage-driven content caching. In *6th Workshop on Web*, 2012.
- [49] M. Marlinspike. New tricks for defeating ssl in practice. *BlackHat*, 2009.
- [50] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. <http://goo.gl/BZXiww>.
- [51] M. Prandini, M. Ramilli, W. Cerroni, and F. Callegati. Splitting the https stream to attack secure web connections. *IEEE Security and Privacy*, 8(6), 2010.
- [52] R. Saltzman and A. Sharabani. Active man in the middle attacks. *OWASP AU*, 2009.
- [53] B. Sterne and A. Barth. Content security policy 1.0. *W3C Candidate Recommendation CR-CSP-20121115*.
- [54] J. Sunshine, S. Egelman, H. Almuhiemedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning

effectiveness. In *USENIX Security Symposium*, 2009.

- [55] M. Vallentin and Y. Ben-David. Persistent browser cache poisoning. <http://goo.gl/xKe06G>.
- [56] M. Vallentin and Y. Ben-David. Quantifying persistent browser cache poisoning. <http://goo.gl/Kn9PGB>.
- [57] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Security and Privacy*, 2010.