Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning

Yaoqi Jia [a], Yue Chen [b], Xinshu Dong [c], Prateek Saxena [a], Jian Mao [b,*], Zhenkai Liang [a]

[a] School of computing, National University of Singapore, 13 Computing Drive, COM1 #3-27, Singapore 117417
[b] School of Electronic and Information Engineering, Beihang University, 37 Xueyuan Road, Haidian District, Beijing 100191, China.
[c] Advanced Digital Sciences Center, 1 Fusionopolis Way, #08-10 Connexis North Tower, Singapore 138632

A R T I C L E I N F O

Article history:
Received 9 February 2015
Received in revised form 17 July 2015
Accepted 21 July 2015
Available online 3 August 2015

Keywords:
Browser cache poisoning
Web security
Browser vulnerability
HTTPS
Script injection

A B S T R A C T

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 cached resources with malicious ones. We investigate the feasibility of such attacks on five mainstream desktop browsers and 16 popular mobile browsers. We find that browsers are highly inconsistent in their caching policies for loading resources over SSL connections with invalid certificates. In particular, the majority of desktop browsers (99% of the market share) and popular mobile browsers (over a billion user downloads) are affected by BCP attacks to a large extent. Existing solutions for safeguarding HTTPS sessions fail to provide comprehensive defense against this threat. We provide guidelines for users and browser vendors to defeat BCP attacks. Meanwhile, we propose defense techniques for website developers to mitigate an important subset of BCP attacks on existing browsers without cooperation of users and browser vendors. We have reported our findings to browser vendors and confirmed the vulnerabilities. For example, Google has acknowledged the vulnerability we reported in Chrome's HTML5 AppCache and has fixed the problem according to our suggestion.

1. Introduction

When browsing the web using HTTPS, if a user Alice ignores, or *clicks through*, the browser's SSL warning of an invalid SSL certificate, she exposes her browser sessions to a Man-In-The-Middle (MITM) attack, allowing attackers to intercept communication in the SSL channel. Recent work has measured the click-through rates for SSL warnings, indicating that more than 50% users click through SSL warnings (Akhawe and Felt, 2013; Dhamija et al., 2006; Sunshine et al., 2009). A typical solution is to improve warnings of invalid SSL certificates (Felt et al., 2014; Sunshine et al., 2009). However, even with the knowledge of an invalid certificate, users often temporarily click through the warnings, e.g., to active Internet access in hotels or cafes through a portal with the self-signed certificate (Chen et al., 2009).

In this paper, we study the consequence of clicking through SSL warnings, focusing on the impact on the browser cache. After Alice clicks through one SSL warning, network

* *Corresponding author.* Tel.: + 861082317212.
E-mail addresses: jiayaoqi@comp.nus.edu.sg (Y. Jia), chenyue@ee.buaa.edu.cn (Y. Chen), xinshu.dong@adsc.com.sg (X. Dong), prateeks@comp.nus.edu.sg (P. Saxena), maojian@buaa.edu.cn (J. Mao), liangzk@comp.nus.edu.sg (Z. Liang).

http://dx.doi.org/10.1016/j.cose.2015.07.004
0167-4048/© 2015 Elsevier Ltd. All rights reserved.

attackers can substitute original web application resources (such as JavaScript code and images) with malicious ones via MITM attacks. By setting long-lived cache headers, the malicious resources may be cached in Alice's browser for a long time. These malicious resources in the web cache[1] are shared across all sites visited by the same browser, affecting future browser sessions involving these resources until the cache is cleared. We call this class of attacks *browser cache poisoning (BCP)* attacks. In our study, we classify BCP attack vectors into three types: In *same-origin BCP attacks*, attackers poison one origin's resources once, and persist them over time using browser cache; in *cross-origin BCP attacks*, attackers corrupt one origin's subresources imported from another origin; in *extension-assisted BCP attacks*, attackers poison subresources inserted by browser extensions.

For HTTPS sessions with invalid certificates, which we call *broken HTTPS sessions* in this paper, browsers vary substantially in how they display warnings and in their caching policies. We evaluate BCP attacks on five mainstream desktop browsers (such as Firefox and Chrome, which cover over 99% desktop browser users by market share (A. Technica, 2015)), and 16 popular mobile browsers (such as Android Default Browser and Dolphin, which have more than one billion mobile browser users by download statistics). We find several serious vulnerabilities in how browsers handle SSL warnings. For example, we find that CM browser 5.0.22 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 web page frames using clickjacking techniques (Huang et al., 2012). Further, the 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 invalid certificates, several SSL warnings do not include the site's name and certificate's contents, or only include the name of the top-level URL rather than the target site's actual URL. Browser extensions in Safari and Opera can inject external scripts loaded over HTTP protocols[2] into HTTPS sessions without raising any SSL or mixed-content warnings, unlike in Chrome and Firefox. We also find that 26 Android applications that embed browsers do not display SSL warnings and are vulnerable to BCP attacks. Such browser issues open up the opportunity for BCP attacks.

Browsers provide more than one kind of caches, e.g., the web cache and the HTML5 application cache (AppCache), and enforce different caching policies. For valid HTTPS sessions, all browsers respect the header's directives and cache resources properly. However, for click-through HTTPS sessions, all these browsers but desktop-version Safari cache poisoned resources in the web cache or in the HTML5 AppCache. Caching policies for the HTML5 AppCache are different between Chrome and Safari. Although Safari can prevent cache poisoning through web application resources, HTTP scripts injected into HTTPS sessions by extensions are cached without triggering SSL warnings. We discuss these differences in depth in Section 4, and show that all five desktop browsers and 16 mobile browsers are susceptible to BCP attacks. We have reported our findings to these browser vendors. Browser vendors, including the vendors of Chrome, Safari, Maxthon, and Dolphin, have acknowledged our findings. Google has confirmed our reported bug in AppCache and awarded a bounty for this finding (Chromium, 2010; G. C. Team, 2015). It has also developed a fix for not caching resources over broken HTTPS sessions in AppCache as we suggest in Section 6.

Many existing defense solutions against HTTPS MITM attacks, e.g., Channel ID (Balfanz and Hamilton, 2013), DANE (Hoffman and Schlyter, 2012), CAA (Hallam-Baker and Stradling, 2013), HSTS (Hodges et al., 2012), HPKP (Evans and Palmer, 2011), SISCA (Karapanos and Capkun, 2014), and other best practices (e.g., enabling CSP (W3C, 2015a)) are adopted in real-world systems. Although they prevent a subset of BCP attacks, none of them provide comprehensive protection. We study the Alexa Top 100 websites, and find that only five sites are protected in some browsers through these defense mechanisms. By analyzing 31,377 websites supporting HTTPS out of Alexa Top 1,000,000 sites, we find that only 510 (1.63%) sites have proper protection enabled. Therefore, developers should not rely on these defense mechanisms as a panacea for BCP attacks. We discuss how browser vendors can make their browser caching policies consistent and correct, thereby eliminating the threat from BCP. Meanwhile, we discuss guidelines for users to protect themselves against BCP attacks. As websites cannot impel all browsers to implement proper protections or force all users to use the upgraded browsers, we also propose defense techniques for web developers to mitigate the impact of these attacks without browser vendors' and users' cooperation.

Contributions. Though BCP attacks have been conceptually discussed in previous work (Kuppan, 2010; Saltzman and Sharabani, 2009; Vallentin and Ben-David, 2010), we give the first systematic measurement of the problem in widely-used browsers and websites. We also give an in-depth discussion of defense techniques. In particular, we make the following contributions.

- **Evaluating the susceptibility of desktop and mobile browsers.** Through experiments, we find inconsistency in SSL warnings among browsers, which may cause victim users to click through warnings. The incoherence of browser caching policies makes browsers vulnerable to BCP attacks. We find that five mainstream desktop browsers and 16 popular mobile browsers are susceptible to such attacks. We also find that 26 Android applications that embed browsers do not display SSL warnings and are vulnerable to BCP attacks. Meanwhile, only five sites of Alexa Top 100 sites and 1.63% of 31,377 HTTPS websites of Alexa Top 1,000,000 sites have partial protections.
- **Analyzing existing defense mechanisms and proposing new defense techniques.** We discuss pros and cons of defense mechanisms, 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

[1] In 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.

[2] We refer to such scripts as HTTP scripts in this paper.

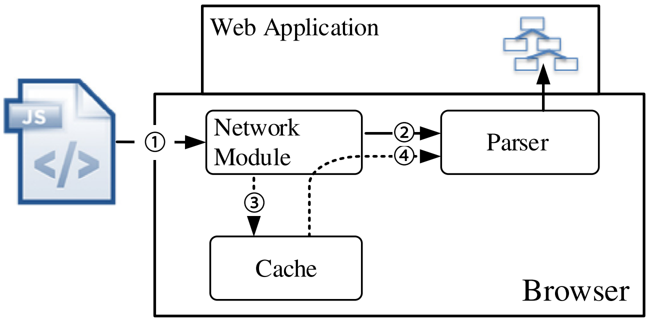


Fig. 1 – Illustration of loading resources in browsers via network (paths 1 and 2 with solid lines) and via cache (paths 3 and 4 with dotted lines).

developers to mitigate cross-origin BCP attacks on the existing browsers.

- **Systematic studying BCP and identifying 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: the extension-assisted BCP attack vector.

2. Background and related work

There are extensive research on attacks on HTTPS/SSL connections and browser cache, as well as the defense solutions. In this section, we first introduce browser cache, and then discuss related work.

2.1. *Background: browser cache*

The main purpose of the browser cache is to reduce the loading time of web pages and resources. Existing browsers employ memory cache and disk cache to store resources, e.g., HTML pages, JavaScript files, CSS files, PDFs, and so on (G. Developers, 2015). When users request such web resources, browsers automatically load the cached resources instead of sending requests to remote servers, as Fig. 1 shows. Caches in browsers mainly include the web cache and the HTML5 application cache. The former is by default active for all web resources, while the latter is an HTML5 feature that needs to be explicitly activated by APIs and configurations in the web application. In this paper, we show that the caching policies implemented in various browsers allow attackers to compromise sessions and make persistent the impact over time, if users click through SSL warnings only once.

Web cache (shared across all sites). The web cache is the default browser cache for all HTTP/HTTPS resources, shared across all sites. Thus, once the browser caches a site's resources over HTTP/HTTPS, if another site requests the same resources, the browser will load the cached copies instead of issuing new requests to remote servers. Version 1.1 of HTTP provides *cache-control* and *expires* headers to specify the expiration time of cached resources, with the former having higher precedence (Fielding et al., 1999). During the specified lifetime, when the cached resources are requested, the browser will not issue any GET request for them until the expiration time or maximum age is reached. Thus in a BCP attack, once the attacker poisons the targeted site's resources in the web cache, the browser will directly load the cached resources for all sites embedding these resources. We will detail the different scenarios for BCP attacks via web cache in Section 3.2.

HTML5 AppCache (dedicated per site). HTML5 introduces a new type of cache, the *HTML5 application cache (AppCache)*. With AppCache, an entire web application can be stored locally, including pages and resources, making the application accessible even without Internet connections (Mozilla, 2015). AppCache requires the web application to include a cache manifest that specifies the resources to be cached. 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. In contrast to the web cache, which is shared across all sites, resources in AppCache can only be accessed by the owner site of the resources. In other words, when a site instructs browsers to store the specified resources in AppCache, browsers only allow the site itself, not other sites, to load the cached resources from AppCache.

2.2. *Related work*

There has been extensive research on attacks to HTTPS/SSL connections and the browser cache, as well as corresponding defenses.

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 the investigation by Akhawe and Felt (2013). 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 (Dhamija et al., 2006; Sunshine et al., 2009). Herzberg studies the basic and advanced indicators and their usability problems (Herzberg, 2009). In addition, Sunshine et al. find that many respondents do not understand SSL warnings, so they simply ignore the warnings (Sunshine et al., 2009). These studies demonstrate that users easily click through SSL warnings. In this paper, we present the consequence after users click through SSL warnings. We show that ignoring warnings can be disastrous to the security and privacy of their web sessions.

Attacks against HTTPS. Prior research has unravelled numerous attacks to compromise HTTPS (Bhargavan et al., 2014; Callegati et al., 2009; Checkoway et al., 2014; Chen et al., 2009; Karapanos and Capkun, 2014; Marchesini et al., 2005; Marlinspike, 2009; Prandini et al., 2010). For example, Karapanos and Capkun (2014) present Man-In-The-Middle-Script-In-The-Browser (MITM-SITB) attacks to bypass enhanced Channel-ID-based defenses. Chen et al. focus on a malicious proxy named "Pretty-Bad-Proxy", which targets browsers' rendering modules above the HTTP/HTTPS layer to void the end-to-end security properties of HTTPS (Chen et al., 2009). Bhargavan et al. report new practical attacks against applications over TLS, which utilize a combination of successive TLS handshakes over multiple connections to disrupt client authentication (Bhargavan et al., 2014). The theoretical analysis and experiments from Checkoway et al. (2014) show that it is practical to exploit the Dual Elliptic Curve Deterministic Random Bit Generator (Dual EC) (N. I. of Standards, 2015) as used in deployed TLS implementations. Prandini et al. (2010) and Callegati et al. (2009) demonstrate practical examples to split the HTTPS stream to attack secure web connections and conduct MITM attacks on the HTTPS protocol. Marchesini et al. conduct a series of experiments and show that client-side SSL is vulnerable to various attacks, e.g., content-only and API attacks (Marchesini et al., 2005). Fahl et al. mount MITM attacks on mobile applications to analyze SSL security in Android (Fahl et al., 2012) and iOS (Fahl et al., 2013).

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 it can persistently compromise the victim's future sessions.

Attacks via browser cache. Felten and Schneider and Bortz and Boneh deploy timing attacks on browser cache to sniff users' browsing histories and steal private information (Bortz and Boneh, 2007; Felten and Schneider, 2000). Wondracek et al. deanonymize social network users by analyzing users' visited URLs (Wondracek et al., 2010). Jia et al. show that timing attacks on browser cache can also be used to infer victim users' geolocations (Jia et al., 2014). On the other hand, researchers have also examined attacks by poisoning web cache, HTML5 AppCache, and other storage (Bursztein et al., 2010; Johns et al., 2013; Klein, 2011; Kuppan, 2010; Lekies and Johns, 2012; Saltzman and Sharabani, 2009; Vallentin and Ben-David, 2010, 2014). For instance, Saltzman and Sharabani study HTTP cache poisoning and categorize these attacks into passive and active attacks (Saltzman and Sharabani, 2009). Vallentin and Ben-David implement a tool called *air-poison* to mount browser cache poisoning via HTTP in wireless networks (Vallentin and Ben-David, 2010). They also quantify the effect of such attacks, and note the conceptual security shortcomings and risks of JavaScript content distribution networks (Vallentin and Ben-David, 2014). Lekies and Johns investigate three attack scenarios, i.e., cross-site scripting, untrustworthy networks and shared browsers, to show how an attacker is able to inject code into web storage (Lekies and Johns, 2012). Kuppan, Klein and Bursztein et al. demonstrate the cache poisoning attack to real-world HTML5 applications on many browsers (Bursztein et al., 2010; Klein, 2011; Kuppan, 2010). Johns et al. show that the HTML5 Offline Application Cache can be misused to conduct reliable DNS rebinding attacks (Johns et al., 2013).

Proxy cache poisoning attacks have been well studied (Huang et al., 2011; Klein, 2011). For example, Klein discusses how to use existing techniques, e.g., HTTP response splitting, to mount poisoning attacks on the reverse proxy and forward proxy (Klein, 2011). Huang et al. conduct experiments to poison the HTTP caches of transparent proxies via socket APIs, which cause malicious contents to be served by the proxy to all of its users (Huang et al., 2011). In this work, we focus on cache poisoning attacks on various browser cache, e.g., the web cache and the HTML5 AppCache.

Although some of the attack vectors discussed in this paper have been 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 sessions in all popular browsers. We further propose novel defense techniques for websites to protect their sessions immediately before browsers might adopt any BCP attack prevention mechanisms in future.

Defenses against HTTPS attacks and browser cache attacks. On the defense side, numerous researchers propose various solutions to protect HTTPS connections from attacks (Balfanz and Hamilton, 2013; Braun et al., 2014; Dacosta et al., 2012; Dahl and Sleevi, 2013; Dietz et al., 2012; Evans and Palmer, 2011; Hallam-Baker and Stradling, 2013; Hodges et al., 2012; Hoffman and Schlyter, 2012; Huang et al., 2014; Jackson and Barth, 2008; Karapanos and Capkun, 2014; W3C, 2015a). 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 (Jackson et al., 2006). Jakobsson and Stamm neutralize browser sniffing by performing URL personalization on the fly at the server side (Jakobsson and Stamm, 2006). Jia et al. (2014) advocate not to cache location-sensitive resources to prevent leaking users' geolocations.

We will make an in-depth discussion of these solutions in Section 5, after describing our findings on BCP attacks. These solutions are not sufficient to prevent all BCP attacks.

3. Problem overview

In this section, we describe the threat model, and define the browser cache poisoning problem.

3.1. *Threat model*

In our threat model, the adversary is a network attacker, who cannot compromise the victim's browser or system. The attacker uses a one-time MITM attack, intercepting the HTTPS connections between Alice's browser and the target site's server only once. The attacker can utilize a host of well-known MITM techniques (e.g., ARP poisoning and DNS pharming attacks) to re-route all of Alice's traffic to himself. To avoid being detected or blocked by security mechanisms, once the attacker completes the one-time MITM attack, he no longer intercepts the traffic from/to Alice.

We assume that the adversary mounts MITM attacks with invalid certificates, such as self-signed certificates or certificates with mismatched domains. These certificates are expected to raise SSL warnings to users.

If attackers compromise CAs to forge certificates, e.g., security breaches of Comodo (2011) and DigiNotar (I. VASCO, 2011), or attack transparent HTTPS proxies using MITM certificates (Huang et al., 2011; Klein, 2011), no browser warnings will be raised to users during MITM attacks. Browsers will cache the malicious resources under the policy for SSL sessions with valid certificates, which is not the focus of our paper. Instead, we study browsers' caching behaviors in HTTPS sessions with invalid certificates.

3.2. *Problem: browser cache poisoning*

Based on scenarios of click-through warnings, we classify the browser cache poisoning attacks into three categories below.

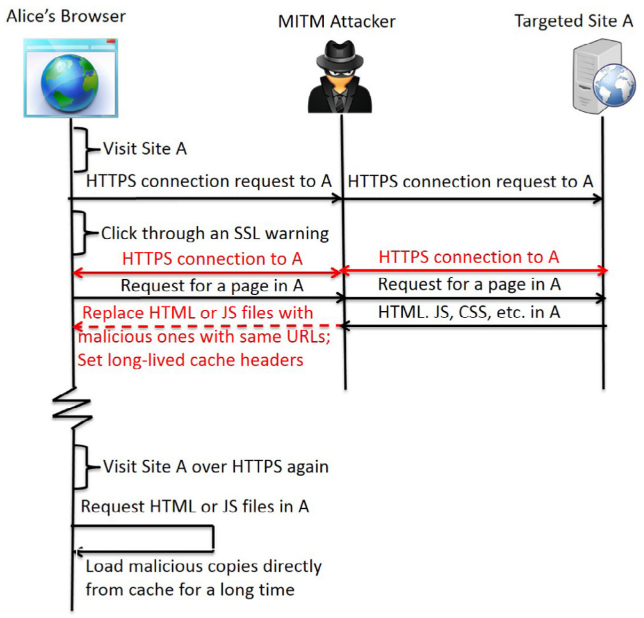

Alice's Browser
MITM Attacker
Targeted Site A
Visit Site A
HTTPS connection request to A
HTTPS connection request to A
Click through an SSL warning
HTTPS connection to A
HTTPS connection to A
Request for a page in A
Request for a page in A
Replace HTML or JS files with malicious ones with same URLs; Set long-lived cache headers
HTML. JS, CSS, etc. in A
Visit Site A over HTTPS again
Request HTML or JS files in A
Load malicious copies directly from cache for a long time

Fig. 2 – Same-origin browser cache poisoning. The attacker conducts the one-time MITM attack while Alice visits the targeted website.

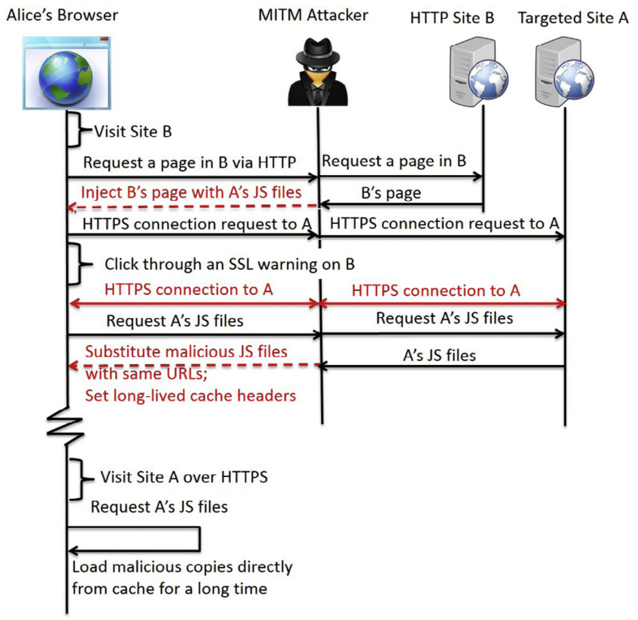

Alice's Browser
MITM Attacker
HTTP Site B
Targeted Site A
Visit Site B
Request a page in B via HTTP
Request a page in B
Inject B's page with A's JS files
B's page
HTTPS connection request to A
HTTPS connection request to A
Click through an SSL warning on B
HTTPS connection to A
HTTPS connection to A
Request A's JS files
Request A's JS files
Substitute malicious JS files with same URLs; Set long-lived cache headers
A's JS files
Visit Site A over HTTPS
Request A's JS files
Load malicious copies directly from cache for a long time

Fig. 3 – Cross-origin browser cache poisoning. While Alice visits an HTTP site B, the attacker injects subresources from the target site A and conducts the one-time MITM attack.

Suppose the targeted site over HTTPS is an online banking website A. The goal of attackers is to impersonate A and cause the victim user's browser to cache malicious copies of resources.

Same-origin. As Fig. 2 illustrates, when an attacker conducts the one-time MITM attack between Alice and the target site A, if Alice clicks through the SSL warning, the attacker can impersonate as the site A, and replace the page and the targeted subresources[3] 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 A's login page. Alternatively, the attacker can utilize HTML5 AppCache to instruct Alice's browser to store the malicious page, manifest, and subresources in the dedicated storage for the banking site for 1 year or longer. Regardless of whether Alice is online or offline, when she revisits the site A, her browser will directly load the whole page from AppCache without issuing any requests. Since the SSL warning occurs on the banking site A, and this attack only affects the same site, we term such attack *same-origin BCP* attack.

Cross-origin. Fig. 3 illustrates the cross-origin attack. Alice is visiting the site B over HTTP. The attacker first intercepts the HTTP connection between Alice and the site B, and injects subresources, say JavaScript, from the site A into B's pages as external JavaScript. Later when Alice's browser sends requests to fetch A's JavaScript, the attacker intercepts the connection and substitutes the returned JavaScript with malicious JavaScript that will be cached for a long time. Since the hijacked banking site uses the certificate forged by the attacker, Alice's browser may raise a warning for the fraudulent certificate on the news site. If Alice clicks through the SSL warning, the malicious JavaScript from the attacker can poison the site A's JavaScript in the browser cache. As a result, the banking site A over HTTPS is compromised when Alice is visiting the news site B over HTTP, even when 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 several websites, all future sessions with these sites are compromised.

Since the SSL warning shown on the site B is for the poisoned subresource, which is a cross-origin resource loaded in the banking site A, we term such attack *cross-origin BCP* attack.

Extension-assisted. The attack targets can be further amplified by browser extensions. Many desktop browser extensions inject resources, e.g., scripts and CSS files, into every page. As Fig. 4 demonstrates, the extension injects JavaScript from its server C into every page. When the attacker conducts the MITM attack to impersonate C, if Alice clicks through an SSL warning for one extension's injected subresource, the attacker can poison the subresource. The consequence is more devastating than previous two scenarios: the poisoned resources will be loaded into every page on which the extension embeds their corresponding JavaScript from the site C. 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 as shown in Fig. 5. After that, if Alice clicks through the warning for mixed contents (Safari and Opera do not have such warnings) on the banking site, the poisoned HTTP resource will be loaded into the banking site, and the site over HTTPS is no longer secure. Since this attack is based on poisoning the extension's injected scripts, we term it *extension-assisted BCP* attack.

[3] The 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.

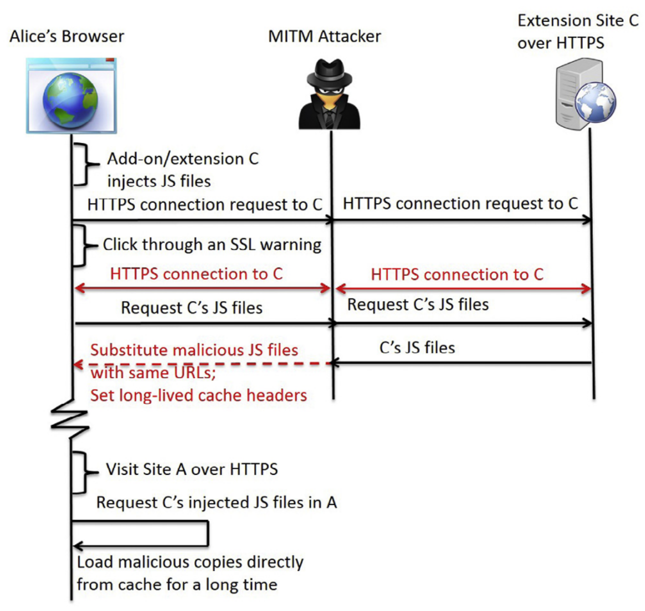

Alice's Browser
MITM Attacker
Extension Site C over HTTPS
Add-on/extension C injects JS files
HTTPS connection request to C
HTTPS connection request to C
Click through an SSL warning
HTTPS connection to C
HTTPS connection to C
Request C's JS files
Request C's JS files
Substitute malicious JS files with same URLs; Set long-lived cache headers
C's JS files
Visit Site A over HTTPS
Request C's injected JS files in A
Load malicious copies directly from cache for a long time

Fig. 4 – Extension-assisted browser cache poisoning. Extensions inject resources for HTTPS sites.

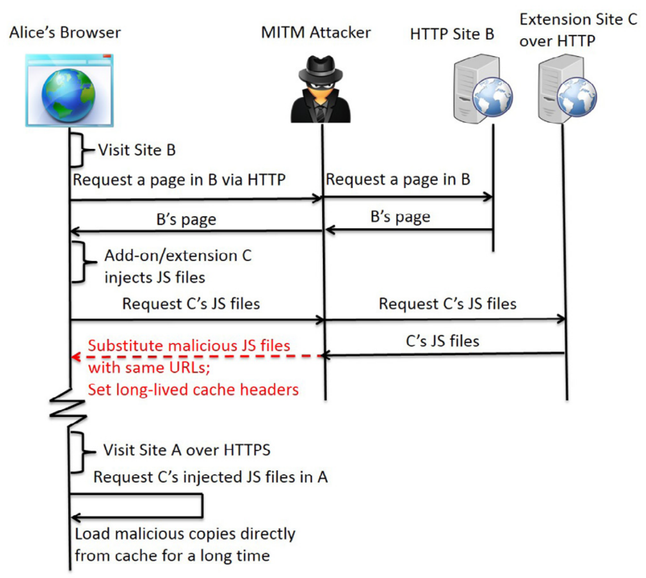

Alice's Browser
MITM Attacker
HTTP Site B
Extension Site C over HTTP
Visit Site B
Request a page in B via HTTP
Request a page in B
B's page
B's page
Add-on/extension C injects JS files
Request C's JS files
Request C's JS files
Substitute malicious JS files with same URLs; Set long-lived cache headers
C's JS files
Visit Site A over HTTPS
Request C's injected JS files in A
Load malicious copies directly from cache for a long time

Fig. 5 – Extension-assisted browser cache poisoning. Extensions inject resources for HTTP sites.

4. Browser measurement

The presentations of SSL warnings and the caching policies vary a lot across browsers. These variances have different impacts on browser cache poisoning attacks. In this section, we make a systematic study of BCP attacks on existing desktop and mobile browsers. We will answer the following questions.

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

We measure BCP attacks on five mainstream desktop browsers and 16 popular mobile browsers. These studies show that all evaluated browsers are susceptible to BCP attacks.

4.1. *Experimentation overview*

We set up an Apache server as the attacker's server, host the malicious resources used in BCP attacks, and use *Cache-Control:public, max-age* = *31,536,000* in the resources' response headers to instruct browsers to cache them for 1 year. We utilize *mitmproxy* (M. dev team, 2014) to intercept the traffic from/to the victim, replace the target 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 (2015) to create self-signed SSL certificates for the target domain.

For the target site, because online banking websites contain users' confidential information, e.g., credit card numbers, these sites are often targeted by attackers. We conduct our experiment on one real-world online banking website. To anonymize the site, we replace the site's name with "OnlineBanking^A" and shade the site's logo in figures throughout the paper (e.g., in Figs. 8, A1 and A2).[4]

We mount BCP attacks on popular browsers (e.g., IE, Chrome, Firefox, Safari, Opera, 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 the mobile browsers have more than one billion downloads. We describe the details of our evaluation below.

4.2. *The inconsistency of SSL warnings*

The SSL/TLS protocol is essential to establish HTTPS connections between servers and browsers.[5] In an MITM attack against HTTPS, the attacker's certificate is not trusted by the victim's browser, which can be caused by certificate with wrong domain name, self-signed certificates, and certificate signed by untrusted CAs. 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.

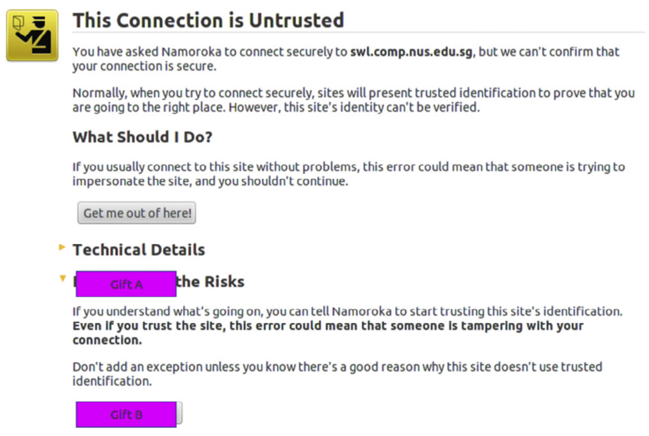
However, as per our evaluation, browsers behave differently in when and how to show such warnings. We gather SSL warnings and address bar warnings on various browsers (shown in Figs. A1 and A2 in the Appendix), and discuss the variances

[4] We denote the targeted site's URL as https://OnlineBankingA/US/JSO/signon/LocaleUsernameSignon.do?locale=en_US, and select the site's essential subresource https://OnlineBankingA/JFP/js/jquery/jquery-1.7.2.js for poisoning.
[5] We use SSL to refer to both TLS and SSL in this paper.

Table 1 – SSL warnings, address bar warnings and default caching policies in mainstream browsers.

	Market share	I	II	III	IV	V	VI	VII	VIII
Desktop browsers									
Firefox (31.0) (Linux, Windows and OS X)	15.54% (A. Technica, 2015)	✓	✓	✓	–	✓	✓	–	–
Chrome (36.0.1985.125) (Linux, Windows and OS X)	19.34%	✓	✓	✓	–	✓	✓	✓	–
Safari (7.0.4) (Windows and OS X)	5.28%	✓	✓	✓	–	✓	✓	✓	
Opera (22.0.1471.70) (Linux, Windows and OS X)	1.05%	✓	✓	✓	–	✓	✓	✓	–
IE (10.0.9200.16540) (Windows)	58.38%	✓	✓	–	–	✓	✓	–	–
Mobile browsers	Number of downloads								
Firefox (31.0) (Android)	50,000,000	✓	✓	✓	–	✓	✓	–	–
Chrome (36.0.1985.125) (Android and iOS)	500,000,000	✓	✓	✓	–	✓	✓	✓	–
Safari (5.0) (iOS)	800,000,000 (Ingraham, 2014)	✓	✓	✓	–	✓	✓	✓	–
Opera (22.0.1485.78487) (Android and iOS)	50,000,000	✓	✓	✓	–	✓	✓	✓	–
IE (10) (Windows Phone)	30,000,000 (A. Research, 2014)	✓	–	✓	–	✓	✓	–	–
Android default browser (4.4.3) (Android)	1,000,000,000 (N. Y. Post, 2014)	✓	✓	–	–	✓	✓	–	–
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	✓	✓	✓	–	–	–	–	–

✓: Yes; –: No.
I : Show pop-up/in-page SSL warnings for sites with invalid certificates.
II: Show address bar warnings for sites with invalid certificates.
III: Block cross-origin subresources with invalid certificates by default.
IV: Show address bar warnings for cross-origin subresources with fraudulent certificates.
V: Display the target site's URL in the SSL warning.
VI: Display the fraudulent certificate's content in the SSL warning.
VII: Not cache resources over broken HTTPS in web cache.
VIII: Not cache resources over broken HTTPS in AppCache.


This Connection is Untrusted

You have asked Namoroka to connect securely to **swl.comp.nus.edu.sg**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

If you understand what's going on, you can tell Namoroka to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

Fig. 6 – The SSL warning inside iframe can be visually overlaid using clickjacking techniques in Firefox 3.6.

in information displayed for resources loaded over broken HTTPS sessions.

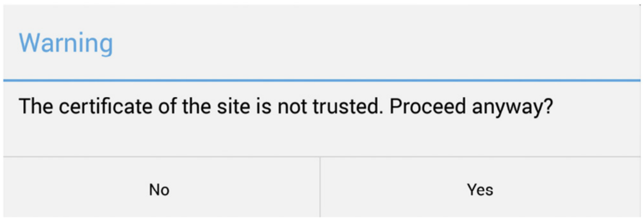
No SSL warnings. CM browser 5.0.22 does not check the validity of certificates, and never shows SSL warnings for fraudulent certificates. It always displays the "Green Shield" in the address bar for all HTTPS connections, regardless of the invalid server-side certificate.[6] This browser has more than ten million users on Google Play.

Overlaid SSL warnings. The attacker can utilize clickjacking (Huang et al., 2012) or tapjacking (Niemietz and Schwenk, 2012) to overlay and camouflage SSL warnings in an iframe, to further lure the victim to click through the warning. Fig. 6 demonstrates the SSL warning inside an iframe can be overlapped and hidden using the clickjacking technique. We have successfully used this technique on Firefox 3.6, IE 8, IE 10 for Windows Phone and other old version browsers.

Incomplete SSL warnings. As Table 1 and Fig. A2 show, the majority of current browsers show SSL warnings for broken HTTPS sessions. For desktop browsers, whereas Firefox, Chrome and IE show in-page SSL warnings for such sites, Safari and Opera display pop-up warnings. For mobile browsers, Firefox, Chrome, Opera, IE and UC display in-page warnings, while other ten 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.

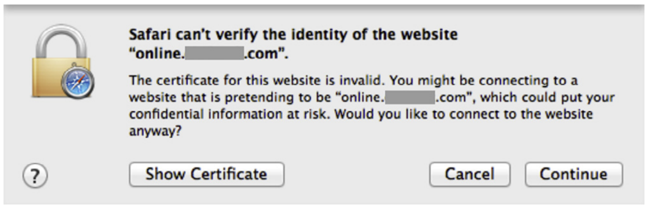
All warnings have the same intended goal, i.e., to alert users that the server's certificate is not trusted, but they have different presentations, e.g., various messages and appearances, as shown in Figs. 7 and 8. We demonstrate the variances of

[6] The vendor adds SSL warnings in the latest version, but CM is still affected by BCP attacks when the user ignores the warning.


Warning

The certificate of the site is not trusted. Proceed anyway?

Fig. 7 – The SSL warning in Javelin displays incomplete information, e.g., missing the target site's URL.


Safari can't verify the identity of the website "online.[redacted].com".

The certificate for this website is invalid. You might be connecting to a website that is pretending to be "online.[redacted].com", which could put your confidential information at risk. Would you like to connect to the website anyway?

Fig. 8 – The SSL warning in Safari shows the target site's URL and the certificate.

the presented information below, and discuss their potential implications.

1) **Different default actions 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 Fig. A2 to caution users. For cross-origin BCP attacks, the attacker can inject subresource from the target site 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 target site are compromised.

2) **Missing URLs in the warning.** As Table 1 shows, except Firefox, Chrome, Safari, Opera, IE (desktop version and mobile version) and Android Default Browser, other ten mobile browsers do not display the target site's URL in the warning. For Baidu, Maxthon, Next, Dolphin, and Boat browsers, after clicking the "View certificate" and "View page info" buttons, the current page's URL (not the hijacked subresource's URL) will be shown. Since the target site'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 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, in the address bar for sites with invalid certificates. For mobile browsers, Baidu, Maxthon, Next, CM, Javelin, Web Explorer Browser, IE for windows phone and Web Browser do not display warnings in the address bar for these sites as shown in Fig. A1n–u, in contrast to other browsers in Fig. A1a–m. For example, CM browser always displays "Green Shield" in the address bar as shown in Fig. A1q. In this case, once the target site is under 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 contents of invalid certificates.** When HTTPS sessions are intercepted, the certificates' contents in the SSL warnings can help users identify whether the certificates should be trusted or not. However, several mobile browsers, e.g., Javelin, Web Explorer, Web Browser, and UC, do not display the certificate's content.

Such inconsistency among today's web browsers in warning users of SSL errors may result in clicking through the warnings. Especially, the improper warning presentations on certain mobile browsers, e.g., no warnings in the address bar and missing the target site's URL, make users more susceptible to BCP attacks. Even worse, when SSL errors are ignored by users, browsers' policies for caching resources are heavily browser-specific. In Section 4.3, we discuss the incoherence of browser policies for broken HTTPS connections.

4.3. *Incoherence of browser caching policies*

Though SSL warnings are inconsistent among browsers and often do not provide enough information to caution users away from target sites, if these browsers employ proper caching policies, they can still 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 HTTPS connections with valid certificates, all browsers will follow the header's directives and cache the resources properly. For broken HTTPS connections after clicking through SSL warnings, different browsers deploy different caching policies.

Caching resources over broken HTTPS in the web cache. As Table 1 shows, only Chrome, Safari, and Opera do not cache resources over broken HTTPS in the web cache, but all other browsers cache the resources. Since the web cache is shared across all sites, if common JavaScript libraries in the web cache are poisoned by BCP attacks, all the sites that contain the same libraries are affected.

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.

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 the web cache or AppCache.

We have reported the incoherence of browser caching policies to the related browser vendors. Google acknowledged these findings with a bounty and is deploying a fix in Chrome suggested by us, as we will propose in Section 6.

4.4. *Susceptibility of browsers*

Same-origin. As Table 1 demonstrates, all browsers cache resources over broken HTTPS in either the web cache or the HTML5 AppCache, except Safari on the desktop platform. For these browsers, once the victim clicks through one SSL warning

Table 2 – Extensions/add-ons that inject HTTP/HTTPS scripts into every page on Chrome.

Name	I	II	Name	I	II	Name	I	II
Free Smileys & Emoticons	1,598,606	1	friGate-unlock sites	265,191	2	Iminent	2,357,926	1
Lightning Newtab	4,397,781	1	Search Switch	307,889	9	Search All	305,701	9
Slick Savings	470,422	5	Everplex Dark	19,676	8	Video download helper	467,430	9
3Dnator	62,861	1	Pacman	82,654	5	Mini Clock	8,243	9
uTorrent for Chrome	89,564	4	PiccShare	94,693	3	Album Downloader	64,415	10
EXIF Viewer	39,674	1	Imageshack-Clickberry	23,020	1	Dailymotion downloader	15,957	3
ShopperPro	305,386	11	Printer button	7164	7	Shopping Helper	575,752	4

I: Number of users (data collected in August, 2014).
II: Number of injected scripts.

on the targeted site, the browser will cache the substituted resources from the attacker, and the site is affected by same-origin BCP attacks.

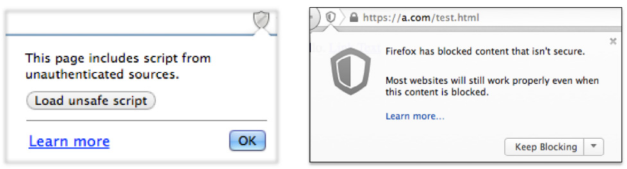
Cross-origin. Table 1 shows that only Chrome, Safari, and Opera do not utilize the web cache to cache resources over broken HTTPS. Furthermore, Firefox, Safari, Chrome, Opera, IE for Windows Phone and UC directly block these fraudulent resources from being loaded into the current page. Thus all other browsers are affected by cross-origin BCP attacks. Because the web cache is shared across different sites, it can be used by such attacks. Once the victim clicks through the warning for the target site's subresource on any HTTP site, the subresources can be replaced by the BCP attacker and cached in the victim's browser for a long time. Thus the warning on other site can affect the target site's security.

Extension-assisted. We observe that many extensions inject resources, e.g., JavaScript, into every page, for example, Free Smileys & Emoticons (1.8M users), Lightning Newtab (4.3M users) on Chrome, and WindowShopper (32,000 users) on Firefox. On Firefox, Chrome, Safari, and Opera, we develop a tool to automatically download extensions and analyze their injections into pages. We show part of our results in Table 2. We mount extension-assisted BCP attacks on these extensions in four browsers and summarize the results in Table 3. Though Table 1 shows that only Firefox caches the injected JavaScript files over broken HTTPS, all these browsers allow extensions to inject HTTP scripts into HTTPS sites and store the malicious HTTP resources in the web cache. Safari and Opera do not display warnings for such mixed contents, 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 contents 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 hand, 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 Fig. 9 shows. Once the victim clicks through the warning, these two browsers will load the subresources over HTTP, which can be affected by BCP attacks.

In conclusion, all the evaluated browsers are susceptible to at least one category of BCP attacks. For desktop browsers, Safari is only affected by extension-assisted BCP attacks, Chrome, Firefox and Opera are vulnerable to same-origin and extension-assisted BCP attacks, while IE is affected by all the three series of attacks. Since all the evaluated mobile browsers do not support extensions/add-ons, they are not susceptible to extension-assisted BCP attacks. For mobile browsers, Firefox, Chrome, Safari, Opera, IE, and UC are only vulnerable to same-origin BCP attacks, and the rest of browsers are affected by both same-origin and cross-origin attacks.

The desktop browsers cover at least 99% of desktop browser users and the mobile browsers have been downloaded over one billion times by mobile users. We conducted our experiments on the browsers with recent versions, and we believe that the old version browsers have the same problems, if not worse.

Susceptible browser-like applications. Mobile platforms (like Android and iOS) provide APIs for developers to embed in-app browsers (e.g., WebView in Android) into their applications. Therefore, it is convenient for users to visit websites inside these browser-like applications. However, since mobile developers are usually not experts in browsers, they may not configure SSL and cache policies for their applications properly, which exposes applications vulnerable to BCP attacks. We conducted BCP attacks on 200 Android applications (from Jin's dataset (Jin et al., 2014)) downloaded from Google Play. We found that 26 of them


This page includes script from unauthenticated sources.
Load unsafe script
Learn more
OK

https://a.com/test.html
Firefox has blocked content that isn't secure.
Most websites will still work properly even when this content is blocked.
Learn more...
Keep Blocking

(a) The warning for mixed content in Chrome. (b) The warning for mixed content in Firefox.

Fig. 9 – The warnings for mixed content in Chrome 36 and Firefox 31.

Table 3 – Caching and mixed content policies for extensions' injected resources on Chrome, Firefox, Safari and Opera.

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

I: Cache extensions' injected resources over HTTP or broken HTTPS.
II: Allow extensions to inject HTTP resources into sites over HTTPS.
✓: Yes; –: No.

Table 4 – Various techniques for mitigating browser cache poisoning attacks.

	I	II	III
HSTS (Hodges et al., 2012)/HPKP (Evans and Palmer, 2011)/DVCert (Dacosta et al., 2012)	✓	–	–
Channel ID (Balfanz and Hamilton, 2013)/SISCA (Karapanos and Capkun, 2014)	–	–	–
DANE (Hoffman and Schlyter, 2012)/CAA (Hallam-Baker and Stradling, 2013)	–	–	–
CSP (W3C, 2015a)	–	–	–
Web Cryptography (Dahl and Sleevi, 2013)/Subresource Integrity (Braun et al., 2014)/CSP 2 (W3C, 2015b)	–	✓	–
Private Browsing Mode (Aggarwal et al., 2010)	–	–	–
Randomization of Resources' URLs (Jakobsson and Stamm, 2006)	–	✓	–
Segregating Browser Cache (Jackson et al., 2006)	–	✓	✓

I: Same-origin browser cache poisoning attacks.
II: Cross-origin browser cache poisoning attacks.
III: Extension-assisted browser cache poisoning attacks.
✓: Mitigate; –: Not mitigate.

are vulnerable to same-origin and cross-origin attacks and do not display SSL warnings when under attack.[7]

Susceptible websites in Alexa Top 100 and Top 1,000,000. Websites cannot completely thwart BCP attacks by themselves, but by enabling proper settings, e.g., HSTS (Hodges et al., 2012), they can partially mitigate such attacks. After our investigation on Alexa Top 100 websites, we find that 51 sites are served over HTTPS, 22 sites set cache headers properly (e.g., no-cache), two sites (i.e., apple.com and bing.com) do not contain cross-origin resources, six sites set CSP headers[8] and five sites[9] enable HSTS headers.

Furthermore, we send HTTPS requests to Alexa Top one million sites to fetch their homepages. We received 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 protection against BCP attacks.

5. 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 the 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 attacks via HTTPS/browser cache can help defend against BCP attacks. However, they are not sufficient. As Table 4 summarizes, CSP (W3C, 2015a), Channel ID (Balfanz and Hamilton, 2013), SISCA (Karapanos and Capkun, 2014), DANE (Hoffman and Schlyter, 2012), CAA (Hallam-Baker and Stradling, 2013), and private browsing mode (Aggarwal et al., 2010) cannot thwart any type of BCP attacks; HSTS (Hodges et al., 2012), HPKP (Evans and Palmer, 2011), and DVCert (Dacosta et al., 2012) can mitigate same-origin BCP attacks; Web Cryptography API (Dahl and Sleevi, 2013), Subresource Integrity (Braun et al., 2014), CSP 2 (W3C, 2015b) and randomization of URLs (Jakobsson and Stamm, 2006) prevent cross-origin BCP attacks; segregating browser cache (Jackson et al., 2006) 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.

5.1. *Defenses against MITM attacks*

Strict transport security (HSTS), public key pinning (HPKP) and direct validation of certificates (DVCert). HSTS (Hodges et al., 2012) is the successor of ForceHTTPS (Jackson and Barth, 2008), which is proposed to mitigate SSL stripping attacks.[10] It provides an HTTP response 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 the HSTS-compliant browsers give users the option to ignore SSL certificate warnings. However, for HSTS, browsers must first

[7] The vulnerable applications are: edu.jhu.idcs.mobile.jcard, eu.weblore.bridgetag.severn, hideshi.y.exoticA, iazresources.com.datoiraziz, ie.bizapps.onefitness, in.followon.sportentertain, info.yamada_ken1.letswalk, jp.co.cyberagent.girlsup, kagoshima.ayanomahoo, miyazaki.ayanomahoo, mobi.horseracingtips.app, net.buzz_app.buzz, net.copapps.erd, okinawa.team.faith.apps, org.jpn.eqm.eq4m, org.wordproductions.abqhaps, org.wordproductions.letitrise, org.wordproductions.prayerforrefreshing, py.com.documenta.contimovil, se.kristnaskolanoasen.ksskolanoasen, take.soft.heisei, take.soft.holiday, take.soft.seirekiwareki, tw.anddev.aplurk, your.app.name.yamauchi, and your.dash.umedu.

[8] plus.google.com, facebook.com, twitter.com, mail.yandex.ru, pinterest.com and e.mail.ru.

[9] facebook.com, twitter.com, dropbox.com, paypal.com and alipay.com.

[10] In SSL stripping (Marchesini et al., 2005), the MITM attacker intercepts all the traffic from/to the victim. When the victim is being redirected to an HTTPS site from an HTTP site, the attacker intercepts the redirect, acts as the other end of the HTTPS session with the site, and sends the unencrypted version of the content back to the victim. In this way, the attacker issues all the requests over the HTTPS connection with site on the behalf of the victim.

connect to the legitimate websites securely to fetch the authorized certificates before connecting to untrusted networks (Huang et al., 2014). Thus if the BCP attack occurs before the victim connects to the legitimate site, the attacker can still poison the target 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) (Evans and Palmer, 2011) allows websites to specify their own public keys with an 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 (G. Project, 2015; Keeler, 2012). 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 target 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 that use HSTS headers (i.e., github.com and twitter.com), where both sites contain the URL https://www.google-analytics.com/analytics.js without HSTS headers. We find that after poisoning *analytics.js*, when visiting these two sites over valid HTTPS, browsers will load poisoned scripts into these two sites without any warning. For extension-assisted BCP attacks, HSTS/HPKP cannot prevent browsers from loading the hijacked extensions' scripts over HTTP/HTTPS into the target 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.

DVCert (Dacosta et al., 2012) allows web applications to directly vouch for the authenticity of its certificates. Based on a modified PAK (Boyko et al., 2000; MacKenzie, 2002) protocol, the browser learns the adequate information to locally verify all the certificates that will be used during a session within the application. This solution can help browsers to detect MITM attacks on the sessions for the deployed site. Therefore, it mitigates the same-origin BCP attacks. However, if the site contains cross-origin subresources that are not protected by DVCert, these resources can still be poisoned by cross-origin BCP attacks. Therefore, DVCert cannot mitigate cross-origin/extension-assisted BCP attacks. Meanwhile, DVCert can only protect sites where the user has an account and a shared secret, and cannot be used to protect the first connection to a website like HSTS. Currently, this solution only has a proof-of-concept extension for Firefox.

Channel ID and SISCA. Channel ID (Balfanz and Hamilton, 2013) is a TLS extension, which was originally proposed as Origin-Bound Certificates (OBCs) (Dietz et al., 2012). Channel ID enables browsers to generate self-signed certificate to conduct TLS client-side authentication, and further prevent MITM attackers to impersonate as the victims' browsers. Server Invariance with Strong Client Authentication (SISCA) (Karapanos and Capkun, 2014) combines Channel-ID-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 ID/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 latest version on the server. Nevertheless, these settings are in response headers, which can be easily replaced by the BCP attacker when poisoning the target resources by setting long-lived cache headers. The attacker can also poison cross-origin subresources or extension's injected resources in the target site. Therefore, when the user visits the target 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 and CAA. The Certification Authority Authorization (CAA) DNS Resource Record (Hallam-Baker and Stradling, 2013) 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) (Hoffman and Schlyter, 2012) enables the administrators of domain names to sign SSL certificates for websites on their domains. Nevertheless, these approaches are based on DNS Security Extensions (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.

5.2. *Content restriction and document integrity*

Content security policy (CSP). CSP (W3C, 2015a) 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 (Sterne and Barth, 2012). Thus CSP does not interfere with extension-assisted BCP attacks. Summarizing, CSP does not prevent BCP attacks.

Web cryptography API and subresource integrity and CSP2. Web Cryptography API (Dahl and Sleevi, 2013) provides a JavaScript API for performing basic cryptographic operations in web applications, for example, encryption, decryption, hashing, and signature generation. Subresource Integrity (Braun et al., 2014) introduces a mechanism for browsers to verifythatsubresources in web applications have been delivered without unexpected manipulation. Subresource Integrity (Braun et al., 2014) extends several HTML elements with an *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-
        YXjDzeQL5UbgNQJj1KoCAA=?ct=application/
        javascript'>
</script>
```

Similarly, Content Security Policy Level 2 (CSP 2) (W3C, 2015b) provides a *hash* attribute for developers to whitelist a particular inline script. Browsers will verify the integrity of the script with the hash attribute before executing it. Web Cryptography API, Subresource Integrity and CSP 2 all 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, recompute the hash value and set the new one in the target site. For extension-assisted BCP attacks, the injected scripts from extensions are beyond the control of these two techniques. Thus Web Cryptography API, Subresource Integrity and CSP 2 cannot defeat these two attacks. These three techniques are still in W3C working drafts, which are not supported by any browser at this moment.

5.3. *Defenses via browser cache*

Private browsing mode. Private browsing modes, such as Private Browsing in Safari/Firefox and Incognito Mode in Chrome, prevent browsers from permanently storing any cookies, histories, caches or other site related states. However, in the private browsing mode, browsers still cache resources of different websites (Aggarwal et al., 2010). Browsers only clear the cached resources after closing windows by users. Thus when browsers are in the 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 the 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: www.google-analytics.com/analytics.js?19991. Thus when a user visits the target 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 and Stamm neutralize attacks via browser cache by means of URL personalization with this idea (Jakobsson and Stamm, 2006). As users cannot predict all the URLs, the target 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 engines. Since the attacker cannot predict URLs of the target site's subresources, cross-origin cannot work. Nevertheless, in same-origin BCP attacks, the attacker can substitute a malicious page for the target 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 the Same-Origin Policy on browser cache to prevent websites from loading cached resources from other sites (Jackson et al., 2006). This approach prevents hijacked resources from being shared across different sites, and every site can only load its own cached resources. This technique thwarts cross-origin and extension-assisted BCP attacks, but not same-origin BCP attacks. In addition, this defense introduces significant performance overhead (Jia et al., 2014).

6. Our defense techniques

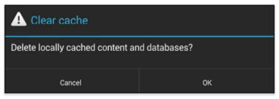
In this section, we first discuss guidelines for users and browser vendors to defeat BCP attacks. However, user faults and browser implementation errors are the main reasons for BCP attacks. We then propose defense techniques for web developers to mitigate cross-origin BCP attacks with minor performance overhead without additional cooperation from browsers and users.

6.1. *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., the web cache and HTML5 AppCache, before visiting a site processing sensitive information, especially after an SSL warning is clicked.

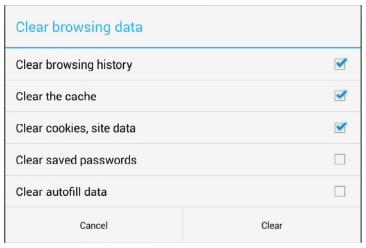
After investigating the settings of 21 browsers, we find that Javelin, Web Explorer and Web Browser do not provide the option for users to clear cache. Safari (mobile and desktop version), IE (Windows Phone version), Android Default Browser and Maxthon have the "Clear cache" button as shown in Fig. 10a, 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 Fig. 10b. 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.

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 to protect themselves from BCP attacks.


Clear cache

Delete locally cached content and databases?

(a) “Clear cache” setting in Android Default Browser.


Clear browsing data

Clear browsing history	✓
Clear the cache	✓
Clear cookies, site data	✓
Clear saved passwords	☐
Clear autofill data	☐

(b) “Clear browsing data” setting in Chrome for Android.

Fig. 10 – “Clear cache” setting in Android Default Browsers does not specify the web cache and HTML5 AppCache. “Clear browsing data” setting in Chrome provides various options for users to clear browsing data.

6.2. *Guidelines for browser vendors*

From the perspective of a browser vendor, to completely defeat BCP attacks, there are two requirements that suffice: (1) Not caching resources over broken HTTPS in either web cache or AppCache; (2) preventing HTTPS sites from loading resources over HTTP by default.

The first requirement protects web 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 contents by default with warnings as shown in Fig. 9, 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.

6.3. *Defense techniques for website developers*

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 browsers. Thus websites need to protect users from BCP attacks even without cooperation of users and browser vendors.

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.

```
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

As discussed in Section 5, five defense techniques – Web Cryptography API, Subresource Integrity, CSP 2, segregating browser cache, and randomization of all resources' URLs – can mitigate cross-origin BCP attacks. However, the first four defenses currently are not deployed in browsers, requiring 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 modification.

Approach overview. Our main goal is to prevent the target site from loading the poisoned JavaScript subresources in the user's browser cache. As Fig. 11 illustrates, in our approach, the target 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 target site's page.

For any subresource B included in the target site's page, we utilize external scripts with a random string in the URL to follow

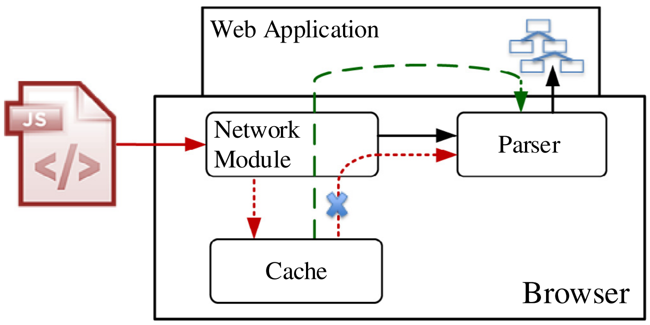


Fig. 11 – Illustration of the defense we propose, which ensures only fresh and unpoisoned subresources can be loaded into the targeted site's page.

the procedure in Algorithm 1. By sending two XMLHttpRequests for B, the 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 passes the check, which indicates B is not poisoned, the scripts append B with the original URL into the page; otherwise, B is poisoned, and thus the scripts append B with B's URL and a random string into the page, which triggers the browser to fetch the latest version of B from the server. Since the browser never loads the poisoned subresources into the target site's page, this approach mitigates cross-origin BCP attacks.

Implementation. Suppose the target web application is a website A with the domain a.com over HTTPS. For infrequently changed resources, e.g., common JavaScript 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. We append our guarding scripts into every page with a random string in the URL, which will not be cached in the client-side browser and follows CSP directives of the page.

(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, our guarding scripts first send *XMLHttpRequest* to fetch them from either the server or the browser cache. Then the guarding scripts 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 the same for two rounds, the subresource is considered to be cached in the user's browser.

Below is the piece of code to measure the load time of *XMLHttpRequest*.

```
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;
    ......
}
```

Listing 1: Measuring the load time for XMLHttpRequests

(2) Based on the caching status of each JavaScript subresource, we have different ways 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, which is used to check the integrity of subresources. 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);
```

Listing 2: Handling different cache and integrity status for one subresource

(3) By default the client-side scripts in A can only issue *XMLHttpRequest* to fetch A's resources, not the resources from other domains. Although the "Access-Control-Allow-Origin" header loosens 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 overcome this challenge, by setting a *reverse proxy* at the server side, 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.[11] As the result of this setting, the URL https://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 *XMLHttpRequest*. 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 target site A, all JavaScript subresources can be classified into four categories: never cached resources with URLs containing a random string, not cached resources with normal URLs, cached resources passing the integrity check with normal URLs, and cached but poisoned resources 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.

[11] 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>.

Table 5 – : Page load time for the original login page and the modified one (in milliseconds) with browser cache.

Website	Time (original)	Time (modified)	Overhead
google.com	779.0	810.5	4.04%
facebook.com	467.0	487.8	4.45%
youtube.com	2134.0	2235.8	4.77%
yahoo.com	1523.0	1587.0	4.20%
twitter.com	331.0	346.9	4.80%
linkedin.com	1340.0	1387.0	3.51%
dropbox.com	1225.0	1265.9	3.34%
paypal.com	548.5	574.1	4.67%
github.com	723.6	752.7	4.02%
wordpress.com	1652.0	1712.5	3.66%

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 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 minor performance overhead (<5%) to these websites. Different from randomizing all resources' URLs, we only randomize the poisoned resource URLs and the browser can still load unpoisoned resources from the cache. Thus our approach causes minor performance overhead.

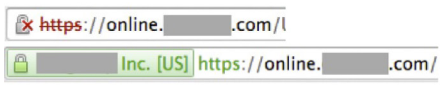
7. Conclusion

In this paper, we perform a systematic study of browser cache poisoning attacks against HTTPS connections, which persistently compromise the victim's web sessions with the target 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 do not deploy SSL warnings properly, and always cache resources over broken HTTPS. In our evaluation, we demonstrate that all 21 popular browsers are susceptible to BCP attacks. We also find that 26 Android browser-like applications do not display SSL warning and are vulnerable 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 web developers to mitigate the impact of these attacks on the existing deployment of browsers.

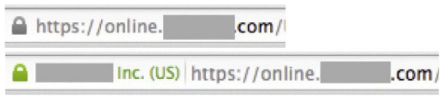
Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 61402029), the National Key Basic Research Program (NKBRP) (973 Program) (No. 2012CB315905), the National Natural Science Foundation of China (No. 61730190), the Beijing Natural Science Foundation (No. 4132056), Ministry of Education – Singapore under NUS grant R-252-000-539-112, and the research grant for the Human-Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR).

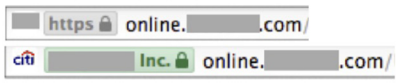
Appendix


~~https~~://online.[redacted].com/
[redacted] Inc. [US] https://online.[redacted].com/

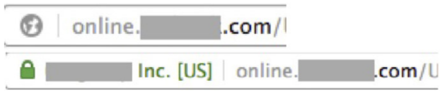
(a) Address bar warnings in Chrome.


https://online.[redacted].com/
[redacted] Inc. (US) https://online.[redacted].com/

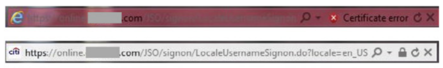
(b) Address bar warnings in Firefox.


[redacted] https online.[redacted].com/
citi [redacted] Inc. online.[redacted].com/

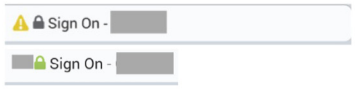
(c) Address bar warnings in Safari.


online.[redacted].com/
[redacted] Inc. [US] online.[redacted].com/U

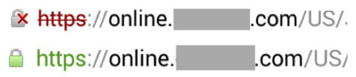
(d) Address bar warnings in Opera.


[redacted].com Certificate error
https://online.[redacted].com/JSO/signon/LocaleUsernameSignon.do?locale=en_US

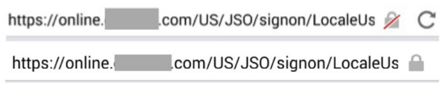
(e) Address bar warnings in IE.


Sign On - [redacted]
[redacted] Sign On - [redacted]

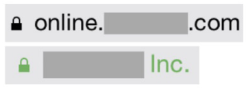
(f) Address bar warnings in Firefox for Android.


~~https~~://online.[redacted].com/US/.
https://online.[redacted].com/US/

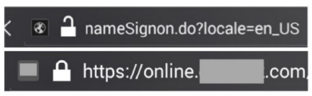
(g) Address bar warnings in Chrome for Android.


https://online.[redacted].com/US/JSO/signon/LocaleUs
https://online.[redacted].com/US/JSO/signon/LocaleUs

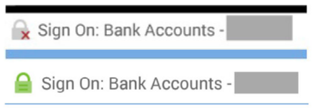
(h) Address bar warnings in Opera for Android.


online.[redacted].com
[redacted] Inc.

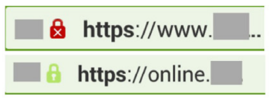
(i) Address bar warnings in Safari for iOS.


nameSignon.do?locale=en_US
[redacted] https://online.[redacted].com

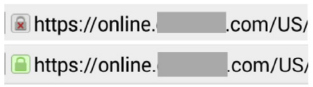
(j) Address bar warnings in Android Default Browser.


Sign On: Bank Accounts - [redacted]
Sign On: Bank Accounts - [redacted]

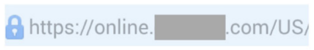
(k) Address bar warnings in UC Browser.


[redacted] https://www.[redacted]..
[redacted] https://online.[redacted]

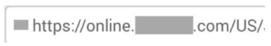
(l) Address bar warnings in Dolphin.


https://online.[redacted].com/US/
https://online.[redacted].com/US/

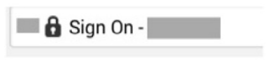
(m) Address bar warnings in Boat.


https://online.[redacted].com/US/

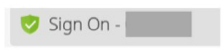
(n) Address bar warnings in Baidu Browser.


https://online.[redacted].com/US/.

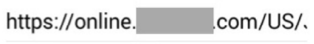
(o) Address bar warnings in Maxthon Browser.


[redacted] Sign On - [redacted]

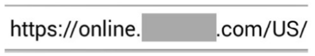
(p) Address bar warnings in Next Browser.


Sign On - [redacted]

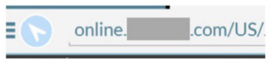
(q) Address bar warnings in CM Browser.


https://online.[redacted].com/US/.

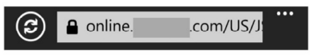
(r) Address bar warnings in Web Explorer Browser.


https://online.[redacted].com/US/

(s) Address bar warnings in Web Browser.

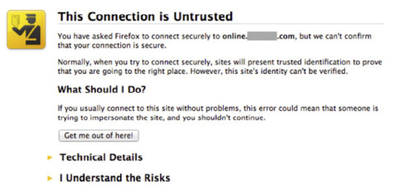

online.[redacted].com/US/.

(t) Address bar warnings in Javelin Browser.


online.[redacted].com/US/J

(u) Address bar warnings in IE for Windows Phone.

Fig. A1 – Warnings in the address bar on mainstream browsers. a–m show warnings for hijacked sites in the address bar: the upper one is the warning for sites over broken HTTPS and the lower one is for sites over secure HTTPS. Other browsers do not show any differences in the address bar for sites with invalid certificates, as shown in n–u.



This Connection is Untrusted

You have asked Firefox to connect securely to **online.[redacted].com**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

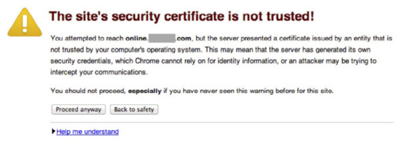
If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Get me out of here!

▸ Technical Details

▸ I Understand the Risks

(a) SSL warnings in Firefox.



The site's security certificate is not trusted!

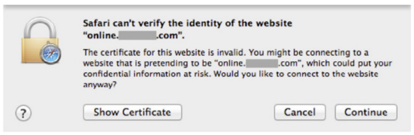
You attempted to reach **online.[redacted].com**, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Proceed anyway | Back to safety

▸ Help me understand

(b) SSL warnings in Chrome.

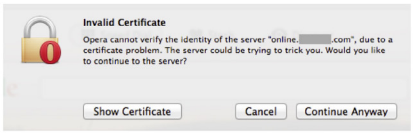


Safari can't verify the identity of the website "online.[redacted].com".

The certificate for this website is invalid. You might be connecting to a website that is pretending to be "online.[redacted].com", which could put your confidential information at risk. Would you like to connect to the website anyway?

? | Show Certificate | Cancel | Continue

(c) SSL warnings in Safari.

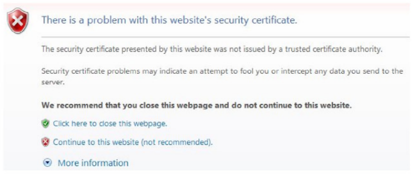


Invalid Certificate

Opera cannot verify the identity of the server "online.[redacted].com", due to a certificate problem. The server could be trying to trick you. Would you like to continue to the server?

Show Certificate | Cancel | Continue Anyway

(d) SSL warnings in Opera.



There is a problem with this website's security certificate.

The security certificate presented by this website was not issued by a trusted certificate authority.

Security certificate problems may indicate an attempt to fool you or intercept any data you send to the server.

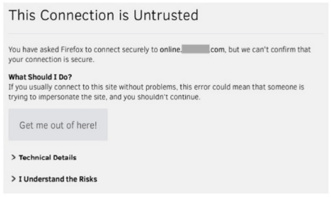
We recommend that you close this webpage and do not continue to this website.

Click here to close this webpage.

Continue to this website (not recommended).

More information

(e) SSL warnings in IE.



This Connection is Untrusted

You have asked Firefox to connect securely to **online.[redacted].com**, but we can't confirm that your connection is secure.

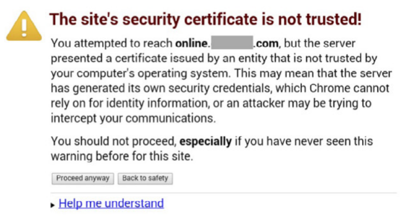
What Should I Do?
If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Get me out of here!

> Technical Details

> I Understand the Risks

(f) SSL warnings in Firefox for Android.



The site's security certificate is not trusted!

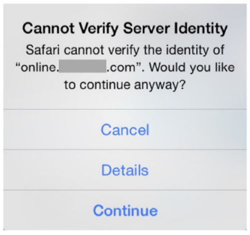
You attempted to reach **online.[redacted].com**, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Proceed anyway | Back to safety

▸ Help me understand

(g) SSL warnings in Chrome for Android.



Cannot Verify Server Identity

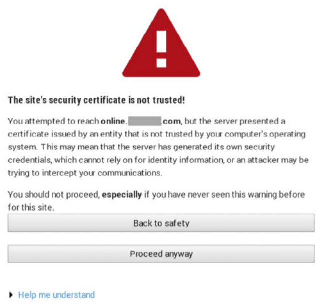
Safari cannot verify the identity of "online.[redacted].com". Would you like to continue anyway?

Cancel

Details

Continue

(h) SSL warnings in Safari for iOS.



The site's security certificate is not trusted!

You attempted to reach **online.[redacted].com**, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which cannot rely on for identity information, or an attacker may be trying to intercept your communications.

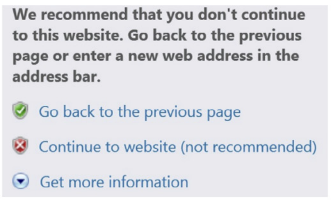
You should not proceed, **especially** if you have never seen this warning before for this site.

Back to safety

Proceed anyway

▸ Help me understand

(i) SSL warnings in Opera for Android.



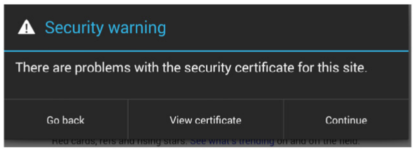
We recommend that you don't continue to this website. Go back to the previous page or enter a new web address in the address bar.

Go back to the previous page

Continue to website (not recommended)

Get more information

(j) SSL warnings in IE for Windows Phone.

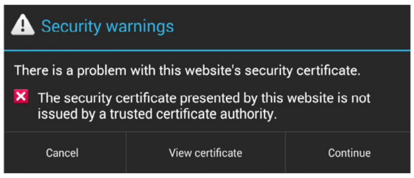


Security warning

There are problems with the security certificate for this site.

Go back | View certificate | Continue

(k) SSL warnings in Android Default Browser.



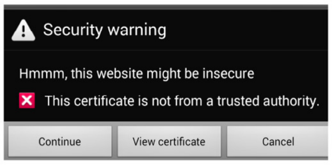
Security warnings

There is a problem with this website's security certificate.

The security certificate presented by this website is not issued by a trusted certificate authority.

Cancel | View certificate | Continue

(l) SSL warnings in Baidu Browser.



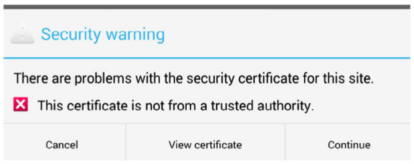
Security warning

Hmmm, this website might be insecure

This certificate is not from a trusted authority.

Continue | View certificate | Cancel

(m) SSL warnings in Maxthon Browser.



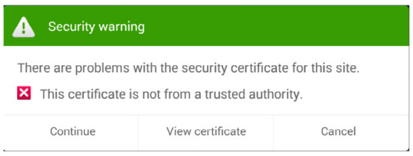
Security warning

There are problems with the security certificate for this site.

This certificate is not from a trusted authority.

Cancel | View certificate | Continue

(n) SSL warnings in Next Browser.



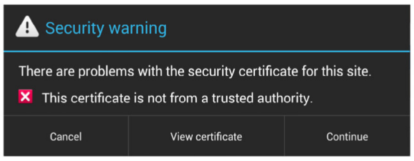
Security warning

There are problems with the security certificate for this site.

This certificate is not from a trusted authority.

Continue | View certificate | Cancel

(o) SSL warnings in Dolphin Browser.



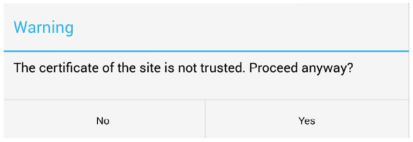
Security warning

There are problems with the security certificate for this site.

This certificate is not from a trusted authority.

Cancel | View certificate | Continue

(p) SSL warnings in Boat Browser.

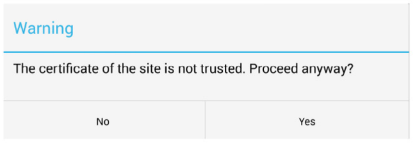


Warning

The certificate of the site is not trusted. Proceed anyway?

No | Yes

(q) SSL warnings in Web Explorer Browser.

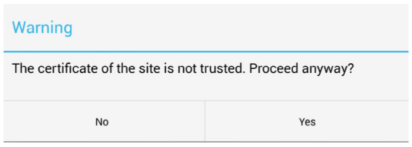


Warning

The certificate of the site is not trusted. Proceed anyway?

No | Yes

(r) SSL warnings in Web Browser.

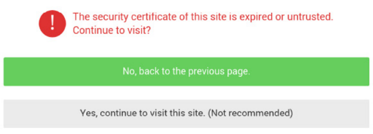


Warning

The certificate of the site is not trusted. Proceed anyway?

No | Yes

(s) SSL warnings in Javelin Browser.



The security certificate of this site is expired or untrusted. Continue to visit?

No, back to the previous page.

Yes, continue to visit this site. (Not recommended)

(t) SSL warnings in UC Browser.

Fig. A2 – SSL warnings in mainstream browsers. As a–p show, SSL warnings in these browsers contain more details of the warning, e.g., certificate. Other browsers display incomplete information in the warning as shown in q–t.

REFERENCES

A. Research. Nokia: 50 million Windows phone sales possible for 2014 (NOK), <http://seekingalpha.com/article/2128173-nokia-50-million-windows-phone-sales-possible-for-2014>; 2014.

A. Technica. Windows 8.x, Internet Explorer both flatline in june, <http://arstechnica.com/information-technology/2014/07/windows-8-x-internet-explorer-both-flatline-in-june/>; 2015.

Aggarwal G, Bursztein E, Jackson C, Boneh D. An analysis of private browsing modes in modern browsers. In: USENIX security symposium. 2010.

Akhawe D, Felt AP. Alice in warningland: a large-scale field study of browser security warning effectiveness. In: USENIX security symposium, 2013. p. 257–72.

Balfanz D, Hamilton R. Transport layer security (TLS) Channel IDs, <https://tools.ietf.org/html/draft-balfanz-tls-channelid-00>; 2013.

Bhargavan K, Delignat-Lavaud A, Fournet C, Pironti A, Strub P-Y. Triple handshakes and cookie cutters: breaking and fixing authentication over TLS. In: 2014 IEEE symposium on security and privacy (SP). 2014. p. 98–113.

Bortz A, Boneh D. Exposing private information by timing web applications. In: Proceedings of the 16th international conference on world wide web. 2007. p. 621–8.

Boyko V, MacKenzie P, Patel S. Provably secure password-authenticated key exchange using Diffie-Hellman. In: Advances in cryptology eurocrypt. 2000. p. 156–71.

Braun F, Akhawe D, Weinberger J, West M. Subresource integrity. In: W3C working draft. 2014.

Bursztein E, Gourdin B, Rydstedt G, Boneh D. Bad memories. BlackHat; 2010.

Callegati F, Cerroni W, Ramilli M. Man-in-the-middle attack to the HTTPS protocol. IEEE Secur Priv 2009;7(1):78–81.

Checkoway S, Fredrikson M, Niederhagen R, Green M, Lange T, Ristenpart T, et al., On the practical exploitability of dual EC in TLS implementations. In: USENIX security symposium. 2014. p. 319–35.

Chen S, Mao Z, Wang Y-M, Zhang M. Pretty-bad-proxy: an over-looked adversary in browsers' HTTPS deployments. In:2009 IEEE symposium on security and privacy. 2009. p. 347–59.

Chromium. Do not cache resources retrieved via broken HTTPS in AppCache or service worker, <https://code.google.com/p/chromium/issues/detail?id=414026>; 2010.

Comodo. Comodo report fraudulently issued certificates, <http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html>; 2011.

Dacosta I, Ahamad M, Traynor P. Trust no one else: detecting MITM attacks against SSL/TLS without third-parties. In: Proceedings of the 17th European symposium on research in computer security, Springer; 2012. p. 199–216.

Dahl D, Sleevi R. Web cryptography API. In: W3C working draft. 2013.

Dhamija R, Tygar JD, Hearst M. Why phishing works. In: Proceedings of the 24th ACM conference on human factors in computing systems. 2006. p. 581–90.

Dietz M, Czeskis A, Balfanz D, Wallach DS. Origin-bound certificates: a fresh approach to strong client authentication for the web. In: USENIX security symposium. 2012. p. 317–31.

Evans C, Palmer C. Public key pinning extension for HTTP, <http://tools.ietf.org/html/draft-ietf-websec-key-pinning-19>; 2011.

Fahl S, Harbach M, Muders T, Baumgärtner L, Freisleben B, Smith M. Why eve and mallory love android: an analysis of android SSL (in) security. In: Proceedings of the 2012 ACM conference on computer and communications security. 2012. p. 50–61.

Fahl S, Harbach M, Perl H, Koetter M, Smith M. Rethinking SSL development in an appified world. In: Proceedings of the 20th ACM conference on computer and communications security. 2013. p. 49–60.

Felt AP, Reeder RW, Almuhimedi H, Consolvo S. Experimenting at scale with google chrome's SSL warning. In: Proceedings of the 32nd ACM conference on human factors in computing systems. 2014. p. 2667–70.

Felten EW, Schneider MA. Timing attacks on web privacy. In: Proceedings of the 7th ACM conference on computer and communications security. 2000. p. 25–32.

Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, et al., Hypertext transfer protocol–HTTP/1.1, <http://tools.ietf.org/html/rfc2616>; 1999.

G. C. Team. Security fixes and rewards, <http://googlechromereleases. blogspot.com/2015/01/stable-update.html>; 2015.

G. Developers. Leverage browser caching, <https://developers.google.com/speed/docs/best-practices/caching>; 2015.

G. Project. HTTP strict transport security, <https://www.chromium.org/hsts>; 2015.

Hallam-Baker P, Stradling R. DNS certification authority authorization (CAA) resource record, <http://tools.ietf.org/html/rfc6844>; 2013.

Herzberg A. Why johnny can't surf (safely)? attacks and defenses for web users. Comput Secur 2009;28(1):63–71.

Hodges J, Jackson C, Barth A. HTTP strict transport security (HSTS), <http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-04>; 2012.

Hoffman P, Schlyter J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA, Tech. rep., RFC 6698, August 2012.

Huang L, Moshchuk A, Wang HJ, Schecter S, Jackson C. Clickjacking: attacks and defenses. In: USENIX security symposium. 2012. p. 413–28.

Huang L-S, Chen EY, Barth A, Rescorla E, Jackson C. Talking to yourself for fun and profit. In: Web 2.0 security and privacy. 2011.

Huang L-S, Rice A, Ellingsen E, Jackson C. Analyzing forged SSL certificates in the wild. In: 2014 IEEE symposium on security and privacy (SP). 2014. p. 83–97.

I. VASCO. Diginotar reports security incident, <https://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx>; 2011.

Ingraham N. Apple has sold more than 800 million iOS devices, 130 million new iOS users in the last year, <http://www.theverge.com/2014/6/2/5772344/apple-wwdc-2014-stats-update>; 2014.

Jackson C, Barth A. Forcehttps: protecting high-security web sites from network attacks. In: Proceedings of the 17th international conference on world wide web. 2008. p. 525–34.

Jackson C, Bortz A, Boneh D, Mitchell JC. Protecting browser state from web privacy attacks. In: Proceedings of the 15th international conference on world wide web. 2006, p. 737–44.

Jakobsson M, Stamm S. Invasive browser sniffing and countermeasures. In: Proceedings of the 15th international conference on world wide web. 2006. p. 523–32.

Jia Y, Dong X, Liang Z, Saxena P. I know where you've been: geoinference attacks via the browser cache. In: Web 2.0 security and privacy. 2014.

Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. Code injection attacks on html5-based mobile apps: characterization, detection and mitigation. In: Proceedings of the 21st ACM conference on computer and communications security. 2014, p. 66–77.

Johns M, Lekies S, Stock B. Eradicating DNS rebinding with the extended same-origin policy. In: USENIX security symposium. 2013. p. 621–36.

Karapanos N, Capkun S. On the effective prevention of TLS man-in-the-middle attacks in web applications. In: USENIX security symposium. 2014. p. 671–86.
Keeler D. Preloading HSTS, <https://blog.mozilla.org/security/2012/11/01/preloading-hsts/>; 2012.
Klein A. Web cache poisoning attacks. In: Encyclopedia of cryptography and security. Springer; 2011. p. 1373.
Kuppan L. Attacking with HTML5. BlackHat; 2010.
Lekies S, Johns M. Lightweight integrity protection for web storage-driven content caching. In: Web 2.0 security and privacy. 2012.
M. dev team. Mitmproxy: a man-in-the-middle proxy, <http://mitmproxy.org/>; 2014.
MacKenzie P. The pak suite: protocols for password-authenticated key exchange. Contributions to IEEE P 1363 2002;2.
Marchesini J, Smith SW, Zhao M. Keyjacking: the surprising insecurity of client-side SSL. Comput Secur 2005;24(2):109–23.
Marlinspike M. New tricks for defeating SSL in practice. BlackHat; 2009.
Mozilla. Using the application cache, <https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache>; 2015.
N. I. of Standards. Technology, Special publication 800-90: recommendation for random number generation using deterministic random bit generators, <http://csrc.nist.gov/publications/PubsSPs.html#800-90A>; 2015.
N. Y. Post. Google: 1 billion people using Android devices, <http://nypost.com/2014/06/26/google-shows-off-android-auto-smartwatches/>; 2014.
Niemietz M, Schwenk J. UI redressing attacks on android devices, <https://media.blackhat.com/ad-12/Niemietz/bh-ad-12-androidmarcus_niemietz-WP.pdf>; 2012.
OpenSSL. OpenSSL project, <https://www.openssl.org/>; 2015.
Prandini M, Ramilli M, Cerroni W, Callegati F. Splitting the HTTPS stream to attack secure web connections. IEEE Secur Priv 2010;8(6):80–4.
Saltzman R, Sharabani A. Active man in the middle attacks. In: OWASP AU, 2009.
Sterne B, Barth A. Content security policy 1.0, W3C Candidate Recommendation CR-CSP-20121115 2012.
Sunshine J, Egelman S, Almuhimedi H, Atri N, Cranor LF. Crying wolf: an empirical study of SSL warning effectiveness. In: USENIX security symposium. 2009. p. 399–416.
Vallentin M, Ben-David Y. Persistent browser cache poisoning, <http://www.eecs.berkeley.edu/~yahel/papers/Browser-Cache-Poisoning.Song.Spring10.attack-project.pdf>; 2010.
Vallentin M, Ben-David Y. Quantifying persistent browser cache poisoning, <http://matthias.vallentin.net/course-work/cs294-50-s10.pdf>; 2014.
W3C. Content security policy, <https://w3c.github.io/webappsec/specs/content-security-policy/>; 2015a.
W3C. Content security policy level 2, <http://www.w3.org/TR/CSP2/>; 2015b.
Wondracek G, Holz T, Kirda E, Kruegel C. A practical attack to deanonymize social network users. In: 2010 IEEE symposium on security and privacy (SP). 2010. p. 223–38.

Yaoqi Jia is a Ph.D. student in National University of Singapore. His research interests are web security and mobile security.

Yue Chen is a Master student in Beihang University, China. His research interests are in web security.

Xinshu Dong is a researcher in ADSC, Singapore. His research interests are in web security, and cyber–physical system security.

Prateek Saxena is an assistant professor in National University of Singapore. His research interests include system security, web security, mobile security, and applied cryptography. He received his Ph.D. degree from University of California, Berkeley.

Jian Mao is an assistant professor in Beihang University, China. Her interests include applied cryptography and cloud security, web security, and mobile security. She received her Ph.D. degree from Xidian University, China.

Zhenkai Liang is an associate professor in National University of Singapore. His research interests include system security, web security, and mobile security. He received his Ph.D. degree from Stony Brook University.