Cookies Lack Integrity: Real-World Implications

Xiaofeng Zheng[1,2,3], Jian Jiang[7], Jinjin Liang[1,2,3], Haixin Duan[1,3,4], Shuo Chen[5], Tao Wan[6], and Nicholas Weaver[4,7]

[1]Institute for Network Science and Cyberspace, Tsinghua University
[2]Department of Computer Science and Technology, Tsinghua University
[3]Tsinghua National Laboratory for Information Science and Technology
[4]International Computer Science Institute
[5]Microsoft Research Redmond
[6]Huawei Canada
[7]UC Berkeley

Abstract

A cookie can contain a "secure" flag, indicating that it should be only sent over an HTTPS connection. Yet there is no corresponding flag to indicate how a cookie was set: attackers who act as a man-in-the-midddle even temporarily on an HTTP session can inject cookies which will be attached to subsequent HTTPS connections. Similar attacks can also be launched by a web attacker from a related domain. Although an acknowledged threat, it has not yet been studied thoroughly. This paper aims to fill this gap with an in-depth empirical assessment of cookie injection attacks. We find that cookie-related vulnerabilities are present in important sites (such as Google and Bank of America), and can be made worse by the implementation weaknesses we discovered in major web browsers (such as Chrome, Firefox, and Safari). Our successful attacks have included privacy violation, online victimization, and even financial loss and account hijacking. We also discuss mitigation strategies such as HSTS, possible browser changes, and present a proof-of-concept browser extension to provide better cookie isolation between HTTP and HTTPS, and between related domains.

1 Introduction

The same-origin policy (SOP) is a corner stone of web security, guarding the web content of one domain from the access from another domain. The most standard definition of "origin" is a 3-tuple, consisting of the scheme, the domain and the port number. However, the notion of "origin" regarding cookies is fairly unusual – cookies are not separated between different schemes like HTTP and HTTPS, as well as port. The domain isolation of cookie is also weak: different but related domains can have a shared cookie scope. A cookie may have a "secure" flag, indicating that it should only be presented over HTTPS, ensuring confidentiality of its value against a network man-in-the-middle (MITM). However, there is no similar measure to protect its integrity from the same adversary: an HTTP response is allowed to set a secure cookie for its domain. An adversary controlling a related domain is also capable to disrupt a cookie's integrity by making use of the shared cookie scope. Even worse, there is an asymmetry between cookie's read and write operations involving pathing, enabling more subtle form of cookie integrity violation.

The lack of cookie integrity is a known problem, noted in the current specification [2]. However, the real-world implications are under-appreciated. Although the problem has been discussed by several previous researchers [4, 5, 30, 32, 24, 23], none provided in-depth and real-world empirical assessment. Attacks enabled by merely injecting malicious cookies could be elusive, and the consequence could be serious. For example, a cautious user might only visit news websites at open wireless networks like those at Starbucks. She might not know that this is sufficient for a temporary MITM attacker to inject malicious cookies to poison her browser, and compromise her bank account when she later logs on to her bank site at home.

We aim to understand how could attackers launch cookie inject attacks, and what are the damaging consequences to real-world websites. Our study shows that most websites are potentially susceptible to cookie injection attacks by network attackers. For example, only one site in the Alexa top 100 websites has fully deployed HTTP Strict Transport Security (HSTS) on its top-level domain, a sufficient server-side protection to counter cookie injection attacks by network attackers (Section 3). We also found a number of browser vulnerabilities and implementation quirks that can be exploited by cookie injection attacks (Section 4). Notably, all major browsers, except Internet Explorer (IE), respect the "Set-Cookie" header in a 407-response (i.e., an Authentication Required Response) when configured to use a proxy. Because of this vulnerability, even websites

adopting sufficient HSTS are subject to cookie injection attacks by a malicious proxy.

Our study also shows that current cookie practices have widespread problems when facing cookie injection attacks (Section 5). We demonstrate multiple exploitations against large websites. For example, we show that an attacker can put his Gmail chat gadget on a victim's screen without affecting the victim's use of Gmail and other Google services. We also demonstrate that an attacker can hijack a victim's online deposit to his account, or even deliver the victim's online purchase to his address. Other exploitations include user tracking, cross-site scripting (XSS) attacks against large financial sites embedded in injected cookies, *etc.*.

We have developed a mitigation strategy (Section 6). By modifying how browsers treat secure cookies, it is possible to largely mitigate cookie injection attacks by network attackers. We have also considered possible browser enhancements to mitigate cookie injection from web attackers. We implement our proposals as a proof-of-concept browser extension. A preliminary evaluation does not encounter compatibility issues.

In summary, this work makes the following main contributions:

- We provided an evaluation of potential susceptible websites to cookie injection attacks, including a detailed measurement of full HSTS adoption and an assessment of shared domains used by Content Delivery Networks (CDNs).
- We examined both browser-side and server-side cookie implementation, in which we found several browser vulnerabilities and a number of non-conforming and/or inconsistent implementations that could be exploited in cookie injection attacks.
- We demonstrated the severity and prevalence of cookie injection attacks in the real world. In particular, our exploitations against a variety of large websites show that cookie injection enables complicated interactions among implements, applications, and various known attacks.
- We developed and implemented browser-side enhancements to provide better cookie isolation. Our evaluation showed promising results in compatibility.

Together, this work provides a close-up picture of the cookie integrity problem and the threats of cookie inject attacks. We intend to provide a context for motivating further discussion in research community and industry.

2 Background

2.1 Cookies

Cookies are a browser-side assisted state management mechanism that are pervasively used by web applications [2]. Cookies can be set by either HTTP servers using "`Set-Cookie:`" header or client side JavaScript with a write to "`document.cookie`". A cookie can have five optional attributes: `domain` and `path` specifying the cookie's scope; `expires` stating when it should be discarded; `secure` specifying that it should only be sent over HTTPS connections, and `HTTPOnly` preventing browser-side scripts from reading the cookie. When sending a request to a server, a web browser includes all unexpired cookies whose domains and paths match the requested URL, excluding those marked as `secure` from the inclusion in an HTTP request.

Cookies have two fairly unusual behaviors. First, there is a critical disconnection between cookie storage and reading. Cookies are set and stored as a name/domain/path to value attributes mapping, but only name-value pairs are presented to both JavaScript and web servers. This asymmetry allows cookies with the same name but different domain and/or path scopes to be written into browser; a subsequent reader can read out all same name cookies together, yet cannot distinguish them because the other attributes such as path are not presented in the reading process. Another complication occurs when writing a cookie, the writer can specify arbitrary value for the `path` attribute, not limited by the URL of the writer's context.

Moreover, the security policy for cookies is not as stringent as the classic SOP. In web security, the SOP is the most important access control mechanism to segregate static contents and active scripts from different origins [3]. An origin for a given URL is defined by a 3-tuple: scheme (or protocol), *e.g.* HTTP or HTTPS, domain (or host), and port (not supported by IE (Internet Explorer)). However, the security policy guarding cookies does not provide separation based on either scheme or port but only on domain [2]. In addition, a website can set cookies with flexible domain scopes: 1) not shared (*i.e.*, host-only), 2) shared with its subdomains, or 3) shared with its sibling domains (i.e., using its parent domain as the scope). For the third case, a restriction is enforced by browser to ensure that a cookie domain scope is not "too wide". For example, `www.example.com` can set a cookie with the scope of `.example.com`, but it cannot set a cookie with `.com` as the scope because `.com` is a public top level domain (TLD). Unfortunately, there is no clear definition of whether a domain scope is "too wide" (See Section 3.2).

The combination of the read/write asymmetry and the

lack of domain or scheme segregation implies that a domain cannot protect the integrity of its cookie from an active MITM or a malicious/compromised *related domain* that shares some cookie domain scope with it. There are two forms of cookie integrity violations:

- **Cookie Overwriting.** If a cookie shares the domain scope with a related domain, it can be directly overwritten by that domain using another cookie with the exactly same name/domain/path. Of particular note, although a `secure` cookie can only be read by an HTTPS process, it can be written or overwritten by an HTTP request.
- **Cookie Shadowing.** Alternatively, an attacker with the control of a related domain can intentionally *shadow* a cookie by injecting another one that has the same name, but different domain/path scope. For example, to shadow a cookie with "value=good; domain=www.example.com; path=/; secure", a related domain `evil.example.com` can write a cookie with "value=bad; domain=.example.com; path=/home". Later, when browser issues a request to `https://www.example.com/home`, both cookies match the URL and are included. For most browsers, the cookie header will be "Cookie: value=bad; value=good;". The "good" cookie could be shadowed by the "bad" one if a website happens to prefer the value of "bad" over "good".

  Note while the "good" cookie has a `secure` flag and is sent over HTTPS, it can still be shadowed with a cookie set from an HTTP connection.

2.2 HSTS

HSTS (HTTP Strict Transport Security) allows a server to inform a client to only initiate communications over HTTPS. It was originally proposed by Jackson and Barth to address a number of MITM threats such as cookie sniffing and SSL stripping [18], and is now standardized in RFC6797 as a HTTP response header `Strict-Transport-Security` [15].

The HSTS header requires a `max-age` attribute indicating how long a browser should keep the HSTS policy for that domain. An optional attribute `includeSubDomains` tells a browser to apply the HSTS policy to its all subdomains. After receiving an HSTS header, a conforming browser ensures that all subsequent connections to that domain always take place over HTTPS until the policy expires. Chrome and Firefox also support a preloaded list that contains self-declared websites supporting HSTS. For more information on HSTS, please see [22].

HSTS coverage can often be incomplete. For example, if `example.com` does not specify `includeSubDomains` in its HSTS header, a browser will allow HTTP connection to `foo.example.com`. Worse, even if the HSTS policy of `example.com` specifies `includeSubDomains`, this will not be checked by a browser if a user only visits `bar.example.com` unless the page includes a reference to `example.com`.

2.3 Cookie Injection Attacks

It is a known vulnerability that cookies can be injected by HTTP response into subsequent HTTPS request, and from one domain to another related domain. Johnston and Moore reported such problem in 2004 [19]. Their report already pinpointed the root cause: the loosely defined SOP for cookies. Unfortunately browsers vendors did not fix the problem probably because they were concerned of potential incompatibility issues. In 2008, Evans described an attack called *cookie forcing* that exploits cookie integrity deficiency to overwrite cookies in HTTPS sessions [7]. In 2013, GitHub migrated their domain for hosting users' homepages from `github.com` to `github.io` after they recognized the threat of cookie injection from/to a *shared domain* whose subdomains belong to mutually untrusted users; they described detailed steps of several possible cookie injection exploits and referred to them as *cookie tossing* [11].

The problem was also noted in several more formal publications. Barth *et al.* discussed security implications of cookie overwriting on session initiation [4]. They also proposed a new header `Cookie-Integrity` to provide additional information so that web server can distinguish between cookies set from HTTP and those set from HTTPS. Bortz *et al.* also reviewed the problem and proposed a new header `Origin-Cookie` that guarantees integrity by enforcing a complete 3-tuple SOP [5]. Singh *et al.* referred the difference between the classic SOP and the cookie SOP as inconsistent principal labeling [30]. Both Zalewski's book [32] and the current cookie specification by Barth [2] explained the cookie integrity deficiencies in great detail. We also learned of two technical reports, one from Black Hat EU by Lundeen [23] and the other from Black Hat AD by Lundeen *et al.* [24], that illustrated several subtle attacks initiated by cookie injection.

Although a known threat, previous research fall short of in-depth empirical assessment of its real-world security implications. This work aims to fill this gap. We provide a detailed comparison in Section 7.

3 Threat Analysis

We first present the threat model for cookie injection attacks. For each type of attacker, we analyze its real-world threat. Table 1 gives an overview.

Attacker		Root Cause	Attack Surface	Mitigation
Network Attacker	Active MITM Malicious Proxy	SOP without protocol & complete domain isolation.	Websites and browsers that allow attackers to reply an unencrypted request to a related domain with forged response.	Full HSTS
Web Attacker	Full control of related domain	SOP without complete domain isolation.	Websites using shared domains.	Public suffix list
	XSS on related domain		Websites with compromised related domains.	Out of scope

Table 1: Overview of the threats of cookie injection attacks

3.1 Threat Model

Two classes of attackers can manipulate a target site's cookies: an active network adversary or a remote adversary able to host or inject content on a related domain.

The active MITM attacker (including the classic MITM fully controlling the network and the Man-on-the-Side (*i.e.*, wiretapping and packet-injecting)) can load arbitrary cookies through HTTP into the target's cookie store. The attacker modifies an unrelated HTTP request to create a hidden iframe in a web page. The attacker's iframe then creates a series of HTTP fetches to the target domains, which the attacker responds to with `Set-Cookie` headers to poison the victim's cookie store.

A malicious proxy is at least as powerful as an active MITM in terms of manipulating network traffic. Moreover, because the browser has extra protocol interactions with the proxy, potential logic flaws or implementation bugs might give the malicious proxy additional chances to break in. Chen *et al.* highlighted this threat with a number of logic flaws [6]. Our study also targets this type of issues related to unexpected capabilities for a malicious proxy to inject cookies.

Finally, if an attacker controls a related domain directly, he may launch cookie injection remotely. The attacker does not need full control of the related server, just the ability to host JavaScript. This attacker cannot target arbitrary domains, but can target any other domain under the same "top level" domain.

One key property of all these adversaries is its ability to change state. For example, a victim might only visit her bank from known-good networks, but an attacker can poison the victim's browser when the victim is on an open wireless network. Only later, when the victim has now returned to the "safe" network and visits her bank, does the attack actually affect the victim.

3.2 Attack Surface

Network Attack Surface: The only current protection against an active network attacker requires that the victim's browser *never* issues an unencrypted HTTP connection to a target site or any related domain. This condition holds if 1) the target domain enables HSTS on its *base domain* [1] (*i.e.* the first upper-level domain that is considered "non-public") with the `includeSubDomains` option, which we refer to as *full HSTS*; 2) the browser supports HSTS; and 3) the browser has received the full HSTS policy from the base domain of the target domain.

[1]We learned this term from Kranch and Bonneau's recent HSTS study [22].

	Domain Ranking					
	<10	10-10^2	10^2-10^3	10^3-10^4	10^4-10^5	$>10^5$
Valid HTTPS	7	52	353	2,914	20,548	128,805
Full HSTS	0	1	7	35	212	997

Table 2: Ranking distribution of domains with valid HTTPS and full HSTS.

Unfortunately, the support and adoption of HSTS in the real world is unsatisfactory. First, all current versions of IE, a major browser with considerable marketshare, do not support HSTS (Microsoft announced that its new browser will support HSTS [16]). Second, there is limited adoption of full HSTS among sites. We scanned 961,857 base domains from the Alexa top one million websites and also examined if these domains present in the Chrome's preloaded HSTS list [28]. While we observed 152,679 (15.87%) domains have deployed HTTPS with valid certificates, we only found 1,252 (0.13%) domains have enabled full HSTS. Moreover, most of the full HSTS domains are low ranked domains (see Table 2). A recent study by Kranch and Bonneau also presented a similar total number of full HSTS domains among the Alexa top one million websites [22].

Because of the prevalence of unsafe networks like open wireless networks and the very limited deployment/availability of full HSTS protection, we consider cookie injection by active network attackers a pervasive and severe threat, especially for websites who have deployed HTTPS to prevent active network attackers from launching other possible attacks such as eavesdropping or active script injection, yet have not enabled full HSTS.

Web Attack Surface: Generally, a web attacker might be able to control a related domain in two ways. First, for large websites that all subdomains are used internally, an attacker can fully control one subdomain by compromising its DNS resolution or its hosting server. The attacker can also exploit a XSS vulnerability on a subdomain of a large website. A cookie injection attack can then be launched to target other subdomains.

A greater concern is when a website either hosts user content or shares a domain scope with other possibly untrustworthy sites. This problem is inherent from the

weaker cookie SOP. As we previously discussed in Section 2.1, a domain is allowed to set cookies with wider domain scope as long as the scope is not considered public. Hence, a clear boundary between "public" and "non-public" domain scope is needed to prevent cookie injection from undesired shared cookie domain. However, this is not easy to define and implement clearly. First, many top-level domains (a.k.a., TLDs), especially country code top-level domains (a.k.a., ccTLDs) have their own reserved suffixes such as `.com.cn`, `.co.uk`, which are mostly TLD-specific. Second, many websites use *shared domains* to assign subdomains to their mutually untrusted clients. Such shared domain providers include cloud hosting providers, web hosting providers, blog providers, CDN providers etc. These shared domains should also be considered as non-public in terms of cookie domain scope.

The problem of cookie domain scope boundary is partially remedied by a community effort initiated by Mozilla called "public suffix list", which maintains an exceptional list containing TLDs, TLD-reserved suffixes, and self-declared shared domains [25]. Public information suggests that the list is enforced by major browser vendors including IE, Chrome, Firefox, and Opera, while our own tests confirm that Safari implements this list.

Our study of the public suffix list shows that the public shared domains list still exposes an attack surface for cookie injection. First, we empirically identified 45 shared domains from the Alexa top one million websites, among which only 10 Google domains and 3 non-Google domains are included in the public suffix list. Among the remaining domains, we found at least 4 domains (`sinaapp.com`, `weebly.com`, `myshopify.com`, and `forumotion.com`) allow customized server-side code or browser-side scripts. Websites hosting on these domains are vulnerable to cookie injection attacks.

Another easy-to-miss corner case is shared domains used by CDNs. CDNs commonly assign subdomains or sub-directories of shared domains to their customers. If a website directly uses a shared domain assigned by its CDN provider, and the CDN provider does not handle the shared domain carefully, then the website is subject to cookie injection attacks from malicious customers of the same CDN provider. While websites rarely use shared domains as their main domains, a common practice is to refer static resources (*e.g.*, JavaScript files, images) using shared domain URLs. Although cookies under these resource URLs are usually not processed by server-side code or browser-side scripts, cookie injection attacks could still cause serious consequences. For example, suppose both websites A and B host their static resource files under one shared domain from the same CDN. Website A can inject garbage cookies from the requests to his resource files with specific paths so that the injected cookies will be sent with the requests of resource files to website B. This type of cookie injection attack could cause performance downgrade, bandwidth consumption, and even denial-of-service (DoS) if the amount of injected cookies exceeds the server's header size limitation[2]. In worst case, DoS of a critical resource file like a JavaScript library could break the whole website.

Vendor	Domain	Publis Suffix List?	Vulnerable?
Akamai	akamai.net	No	n/a [1]
	akamaiedge.net	No	n/a
	akamaihd.net	No	n/a
	edgesuite.net	No	n/a
Azure	msecnd.net	No	Yes
	windows.net	No	Yes
BitGravity	bitgravity.com	No	n/a
CacheFly [2]	cachefly.net	No	Yes
CDN77	cdn77.net	No	Yes
CDNetworks	cdngc.net	No	n/a
CDN.net	worldcdn.net	No	n/a
ChinaCache	chinacache.net	No	n/a
ChinaNetCenter	wscloudcdn.com	No	n/a
CloudFlare [3]	cloudflare.net	No	Yes
CloudFront	cloudfront.net	Yes	No
EdgeCast	edgecasecdn.net	No	n/a
Exceda	expresscdn.com	No	Yes
Fastly [3]	fastly.net	Yes	Yes
Highwinds	hwcdn.net	No	n/a
Incapsula	incapsula.net	No	Yes
Internap	internapcdn.net	No	n/a
Jiasule	jiashule.com	No	Yes
KeyCDN [2]	kxcdn.com	No	Yes
Level3	footprint.net	No	n/a
Limelight	linwd.net	No	n/a
MaxCDN	netdna-cdn.com	No	Yes
Squixa	squixa.net	No	n/a

1: "n/a" refers to the case that we were not able to test.
2: CDNs attempting to defend cookie related attacks on shared domains by filtering the `Set-Cookie` header.
3: CDNs allowing shared cookie scopes in customer-specific prefixes of shared domains.

Table 3: Assessment of cookie injection attacks on shared domains used by CDNs.

We empirically collected 28 shared domains used by 23 CDNs [3], in which only 2 domains are registered in the public suffix list, as presented in Table 3. We were also able to sign up and test 13 shared domains from 12 CDNs. While we confirmed that `cloudfront.net` is immune because of its presence in the public suffix list, for each of the other 12 domains, we successfully launched DoS attack on one test URL by injecting 72KB cookies from another test URL. Our experiments also found two problematic behaviors. First, CacheFly and KeyCDN attempt to defend cookie related attacks by filtering the `Set-Cookie` header in response instead of utilizing the public suf-

[2] Although the current HTTP specification does not define any limitation on the size of request header [9], most of web server implementations do so by default. For example, nginx by default limits a single HTTP header not to exceed 8KB [26].

[3] We collected most of the CDNs from `http://www.cdnplanet.com/cdns/`.

fix list, which fails to prevent JavaScript from injecting cookies. Second, although Fastly has declared several subdomains of `fastly.net` as public suffix, its naming mechanism enables shared scopes in customer-specific prefixes, making its customers still vulnerable to cookie injection attacks. For example, for a customer `foo.com`, Fastly assigns a customer subdomain `foo.com.global.prod.fastly.net`. Although the suffix `global.prod.fastly.net` is present in the public suffix list, the prefix causes a cookie scope `com.global.prod.fastly.net` shared with other customer subdomains such as `bar.com.global.prod.fastly.net`. CloudFlare also has the same problem. We have reported this problem to all vulnerable vendors. CloudFlare and CDN77 have acknowledged our reports. The response from CloudFlare said that they are considering to disable direct access of all `cloudflare.net` URLs to defend against this problem.

4 Pitfalls in Cookie Implementations

Based on the threat model and the understanding of potential attack surfaces, we then turn to understand how cookie related mechanisms are implemented in browsers and web applications. Our study pinpointed a number of inconsistent and/or non-conforming behaviors in major browsers and web frameworks, as summarized in Table 4. We also identified several vulnerabilities in major browsers allowing an active network attacker to inject cookies even when the full HSTS is deployed. We have reported these vulnerabilities to browser vendors.

4.1 Uncovered Implementation Quirks

Browser-side Cookie Ordering. The current cookie specification [2] suggests that browsers should rank cookies first by path specificity and then by the creation time in ascending order. We found all major browsers follow this suggestion except Safari, which ranks cookies first by the specificity of the domain attribute then by the path specificity.

Server/script-side Cookie Preference. The cookie header is semantically a list. For the same name cookies in the list, the specification states that the server should not rely upon cookie's ordering presented by the browser. We examined popular web programming languages, web frameworks, and third-party libraries including PHP, Python, Java, Go, ASP, ASP.NET, JavaScript, Node.js, JQuery, JSF, SpringMVC. At the language level, only Java, JavaScript and Go provide built-in or standard library interfaces to read cookies as a list. Other languages, and all web frameworks and third-party libraries treat the cookie list as a name-value map that only returns one value for each cookie name in the list. For cookies with the same name, while the name-value map interface in Python standard library prefers the last-ordered cookie, all others prefers the first-ordered one. This explains why cookie shadowing is possible and the example given in Section 2.1 works in many cases.

Cookie Storage Limitation. The specification has several vague suggestions for browsers to limit the number and size of stored cookies. We found all major browsers set the maximum size of a single cookie to 4 KB. Chrome, Firefox, and Opera implements a cookie jar for every base domain, with the total numbers of cookies limited to 180, 150, and 180, respectively. IE's cookie jar implementation is per cookie domain scope, with the total number of cookies limited to 50. We did not reach Safari's cookie storage limit after writing and reading 1,000 cookies.

Cookie Header Size Limit. While Safari does not seem to have a limit for the number of cookies, it truncates the matching cookie list if the length of the cookie header exceeds 8 KB. We did not observe similar behaviors in other browsers.

Cookie Name. The cookie name can contain all US-ASCII values except control characters and separator characters (see definition in [2] and [8]). We found that Safari mistakenly stores cookie name in case-insensitive manner. Some programming languages also implement cookie names incorrectly. Previously Lundeen *et al.* reported that ASP.NET implements cookie names case-insensitively [24]. We found that ASP makes same mistake. In addition, PHP performs percent-decoding on cookie names. For these languages, different cookie names sent by browser are possibly recognized as same name cookies, which embraces another vector for cookie shadowing. For example, PHP interprets a cookie header "%76alue=bad; value=good;" as "value=bad; value=good;", causing the "good" cookie to be shadowed by the "bad" one.

Cookie Path. According to the specification, a cookie matches a URL only when the path scope of the cookie is a sub-directory of (or identical to) the URL path. When a cookie does not specify the `path` scope, the browser is required to set its default path as the directory-portion of the URL path without any trailing slash. We found 4 violations to the standard: 1) Safari[4] implements a substring other than sub-directory matching rule; 2) Firefox and IE match cookie path with not only the URL path, but also the URL query and the URL fragment portion match; 3) Firefox matches a cookie path with a URL path when the former has one more slash than the later; 4) Chrome, Safari, and Opera (Linux and iOS versions) include the trailing slash in default cookie path.

[4] Also Chrome on iOS, but as iOS browsers need to use Apple's rendering engine rather than their own, this is probably due to Apple's decision, not Google's

Cookie Property	Specification	Non-conforming/inconsistent behaviors
Browser-side priority	Cookies SHOULD be ranked by specificity of path then by creation time in ascending order.	1. Safari ranks cookies by specificity of domain then by specificity of path.
Server/script-side preference	Server SHOULD NOT rely on cookie's ordering presented by browsers.	1. Most standard libs and frameworks only provide name-value map interfaces; 2. For each name in the cookie list, Python prefers the last-ordered cookie, others prefer the first-ordered one.
Cookie storage limitation	Several vague suggestions	1. Safari seemingly does not have limitation on the number of stored cookies; 2. Chrome and Firefox limit the size of the cookie store per base domain, IE does so per specific domain scope.
Cookie header size limitation	Not specified	1. Safari truncates the cookie header not to exceed 8,192 bytes.
Cookie name	US-ASCII values except control characters and separator characters (see definition in [2] and [8])	1. Safari is case-insensitive with cookie name; 2. ASP and ASP.NET are case-insensitive; 3. PHP performs percent-decoding on cookie name.
Cookie path	1. Cookie path and URL path MUST be identical or sub-directory matching; 2. Trailing slash MUST NOT be included in default cookie path.	1. Firefox and IE matches cookie path not only with URL path, but also with URL query and URL fragments; 2. Safari implements sub-string matching other than sub-directory matching; 3. Firefox allows cookie path has one more slash than the URL path; 4. Chrome, Safari, and Opera under some platforms include trailing slash in the default cookie path.

Table 4: Summaries of non-conforming and inconsistent behaviors found in browser and web server cookie implementations.

4.2 Uncovered Vulnerabilities

Vulnerabilities in Handing Proxy Response. In [6], Chen *et al.* found a number of flaws in major browsers. The root problem resided in the handling of HTTPS responses. Essentially, all browsers at that time could not differentiate an HTTPS response from a proxy and an HTTPS response from the intended server. The flaws were patched after disclosure. However, we found the patches are incomplete: if a proxy replies to a HTTPS `CONNECT` request with an unencrypted 407 (proxy authentication required) response, all major browsers except IE accept the cookies set in 407 response. While some vulnerable browsers display a pop-up window, some accept cookies silently (Table 5).

These vulnerabilities allow a malicious proxy to launch cookie injection attacks against a full HSTS site. Users who use proxies or have them set automatically, these vulnerabilities can also be exploited by an active MITM between the victim and the proxy, even if a victim user does not intentionally use the attacker as the proxy.

Vulnerability in Handing Public Suffixes in Safari. As described in Section 3.2, the public suffix list enforces the boundary between public and non-public cookie domain scopes. However we found the implementation of Safari is vulnerable under certain conditions. When Safari issues a request `http://tld/`, it accepts cookies in the response with domain scope as `.tld`, which are shared by all `subdomains.tld`. Because HSTS is not enabled on an entire TLD (in general, there is no A record indicating a server at the TLDs), this vulnerability is exploitable by active network attackers who can forge a DNS response as well as an HTTP response.

Vulnerability in Safari's HSTS Implementation. We also found a vulnerability in Safari's HSTS implementation. When receiving a URL, Safari does percent-decoding and upper-to-lower case conversion on its domain name before issuing a request. However, the HSTS check is performed before the conversion process completes, enabling an attacker to bypass Safari's HSTS check if both capital and percent-encoding are used in the domain name.

	Windows	Mac OS	Linux	Android	iOS
IE	–	N/A	N/A	N/A	N/A
Chrome	①	①	①	①	①
Firefox	②	②	②	②	N/A
Safari	②	③	N/A	N/A	②
Opera	①	①	N/A	①	N/A

①: cookie injection with pop-up window.
②: cookie injection without pop-up window.
③: cookie injection and script injeciton.

Table 5: Browser vulnerabilities in handling 407 response by a malicious proxy.

5 Real-World Exploitations

Our study aims at understanding the prevalence and severity of potential exploitation by cookie injection in real-world websites. In particular, we are curious about how web developers use cookies, whether they are aware of this problem explicitly and have developed best practices accordingly. With these questions in mind, we conducted black box penetration tests on a number of popular websites with our test accounts. We also reviewed several well-known open source web applications. For penetration tests, we first used browser extensions like EditThisCookie [1] to test manually. For possible exploitations, we then implemented with Bro [27] (for packet sniffing and injection with the `rst` tool) in an open wireless network setting.

We found cookie injection attacks are possible with very large websites and popular open source applications

including Google, Amazon, eBay, Apple, Bank of America, BitBucket, China Construction Bank, China UnionPay, JD.com, phpMyAdmin, and MediaWiki, among others. The consequences of attacks include, but are not limited to, XSS, privacy leakage, bypassing of cross-site request forgery (CSRF) defenses, financial loss, and account hijacking. The varieties of vulnerable web applications and exploitations suggest cookie injection is a serious threat in the real world, and deserves a greater attention from the web security community.

The exploitations we found indicate three common cookie usages: 1) using cookies as authentication tokens; 2) associating important and session independent states with cookies; 3) reflecting cookies into HTML. These cookie usages often lead to cookie injection attacks if specific defensive measures are not in place.

We present our exploitations based on these categories, along with the necessary background and additional observations. Please refer Section 4 and Table 4 for the details of different cookie implementations involved in some cases. We extensively make use of cookie shadowing. For these cases, unless otherwise specified, we assume that the web server has the common behavior of preferring the first-ordered cookie for each name in the cookie list.

5.1 Cookies as Authentication Tokens

A common practice in web development is to use a cookie to identify a user session. Many websites further set long expiration durations on session cookies to avoid having users sign in every time. This practice itself is somewhat questionable, because session cookies are sent along with HTTP requests automatically, which facilitate CSRF attacks. Nevertheless, Barth *et al.* showed that CSRF attacks can be defeated with specific defensive principles and techniques in web applications [4].

Also in [4], Barth *et al.* noted a special form of CSRF which they called *login CSRF*. In this attack, an attacker signs in with his own account on the victim's browser. If not noticed, the victim might visit targeted web site on behalf of the attacker's account, resulting in security and privacy consequences such as search history leakage, credit card stealing, and XSS. The authors also pointed out that login CSRF is a special form of a threat they called *Authenticated-as-Attacker*, which can also be carried out by injecting malicious session cookies to overwrite original ones.

In fact, the consequences of cookie injection on session cookies can go beyond those described in [4]. We found that, by using cookie shadowing, similar attacks could be carried out without noticeable evidences by the victim. We call our attacks *sub-session hijacking attacks*.

5.1.1 Exploiting Google Chat and Search

We first present two exploits targeting Google, which lead our observation of the sub-session hijacking attack. Google's base domain `google.com` is not protected with full HSTS, so in most cases it is subject to cookie injection by an active network attacker.

Case-1: Gmail chat gadget hijacking. The web interface of Gmail at `https://mail.google.com/` shows a chat gadget at the bottom left corner. If an attacker hijacks the gadget without affecting Gmail and other Google services, he can fake the victim's friend list and chat with the victim to initiate advanced phishing, intercept communication, or perform other disruptive activity. This could be particularly deceptive in a targeted attack scenario.

We have confirmed this attack. Although the browser displays everything as one page, the chat gadget and Gmail content are actually loaded with different URLs then composed together. Both the chat gadget and Gmail use cookies for authentication. If an attacker injects his Google session cookies in a way that the injected cookies shadow the original ones only at the chat gadget related URLs, then the attacker can put his chat gadget on the victim's screen, without disturbing the victim's use of Gmail and other Google services.

We demonstrated this attack by injecting a total of 25 cookies: five session cookies "SID/SSID/HSID/APISID/SAPISID", each with five specific paths. Meanwhile most Google services are not affected because the specific paths of the injected cookies do not match with their URL paths. This is sufficient to cause the chat window to load with the attacker's cookies, while all other components are loaded as the victim.

Case-2: "Invisible" Google search history stealing. Another attack is to use cookie shadowing to steal Google search history (which is automatically logged and retrievable with the login cookie) without being noticed. We assume that a user has visited `https://www.google.com/`, which shows the search box and her profile name and icon. When she types in the search box, browser-side script issues AJAX requests to `https://www.google.com/search` to get search results.

Our original goal was to only shadow the session cookies of the AJAX request, so that we could steal search history without affecting the web interface loaded by `https://www.google.com/`. But it turned out we could not achieve this. We first injected three relevant session cookies "SID/SSID/HSID" with path "/search". However, this attempt failed because we found the server unusually preferred the last-ordered cookie, and the injected cookies were ranked before the legitimate ones because of the specific path. We then found out a way to only shadow the session cookies of the AJAX re-

quest on Safari by exploiting its cookie header limitation (see Case-5 in Section 5.1.5 for the details). However, the server seemed to check whether session cookies under `https://www.google.com/search` are consistent with those under `https://www.google.com/`. Once receiving inconsistent session cookies from the AJAX request, it navigated the web interface to `https://www.google.com/search`, which still showed the attacker's profile name and icon.

Our final attack was to inject session cookies with domain scope "`www.google.com`" and path "/", so that for non-Safari browsers, the attacker could steal the victim's search history. Although this attack affects the web interface, causing to show the attacker's profile name and icon, it does not affect most other Google services. We also verified an invisible attack by spoofing the victim's profile name and icon.

5.1.2 Sub-session Hijacking Attacks

The two cases above show a common pattern: the attacker intends to limit the effective scope of injected session cookies as small as possible to reduce the visibility of his attack.

Essentially, web applications require one or more request-reply pairs with different URLs, which we view as different *sub-sessions*. In a normal case, when a user views a web page or performs a certain action through a series of pages, the corresponding sub-sessions carrying the same user authentication tokens are attributed to the user's account. However, when using cookies as authentication tokens, the cookie-URL matching rules and implementations often allow the attacker to selectively associate one or more sub-sessions to the attacker's account by cookie shadowing. That is why we call this type of attack sub-session hijacking attacks.

The impact of such attacks varies by the applications. In general, the attacker's strategy is to select a minimum set of sub-sessions that achieve his attack goals meanwhile keep the visibility of the attack as small as possible. However such attack could be made difficult by some implementation choices.

First, in general, a victim could notice a sub-session hijacking attack if she views abnormal changes of some visual elements on her screen. Typically such elements include username, email, a profile icon *etc.*, which we refer to as *ID-indicators*. If a website uses less URLs in one page or one certain functionality, and makes the important URLs related with the ID-indicators, the attacker is less likely to perform sub-session hijacking without being noticed. For example, in Case-2, the attacker has to hijack both of the URL that shows the search interface, and the AJAX request that performs the search. This limitation causes the expose of his profile name and icon, which may be noticed by the victim. However, if the attacker can only hijack an AJAX request which is not related to the interface, especially ID-indicators, the attack could be launched invisibly.

Second, explicit and session dependent verifiers could bind separate URLs together, so that the attacker needs to hijack more URLs. One example is using a session dependent nonce to counter CSRF attacks. Suppose the attacker wants to steal some sensitive information submitted by a form which is fetched from URL `GetForm` then submitted through URL `SubmitForm`. If the CSRF protection of the form is session dependent, *e.g.* a nonce associated with user session embedded in the form and verified when submitting, the attacker must hijack both `GetForm` and `SubmitForm` so that the CSRF verification does not fail. Otherwise he only needs to hijack `SubmitForm`.

It turns out that sub-session hijacking can be a powerful attack against today's websites. Because many web applications do not adopt mechanisms to bind subsessions together, and, for many operations, hijacking one sub-session is sufficient to cause serious consequences. Below we describe three common functionalities that are often vulnerable to sub-session hijacking, demonstrated with real-world cases.

5.1.3 Payment Account Stealing

Many websites require users to associate one or more payment accounts to pay their bills or online purchases. If the attacker hijacks the payment account submission form, he could get sensitive information, or even spend money using the victim's payment account.

Case-3: Credit card stealing on China UnionPay. China UnionPay, a government-owned financial corporation in China, has an online third-party payment service in which users can add their credit/debit cards. Although the process of adding a card involves four URLs as well as authentication via text message, all the URLs merely use one session cookie "uc_s_key" for authentication and the actual data submission is performed at one URL that is not related to any ID-indicator. We have verified that by shadowing the session cookie at the submission URL, the attacker can hijack China UnionPay's credit card association invisibly to obtain the victim's (obfuscated) credit card number and its spending history when the victim uses this interface in the future.

5.1.4 Online Deposit Hijacking

A common feature in many Chinese websites is the ability to deposit money from an online bank (or a third-party payment service like Alipay) into a website for future

spending. We found this feature is particularly vulnerable to sub-session hijacking.

The process of online deposit usually includes six steps: 1) the user enters deposit amount; 2) the website generates an ID as a unique identifier of this transaction; 3) the website redirects the user to the selected online bank with the transaction ID; 4) the user authenticates and confirms to withdraw money from the online bank; 5) the online bank notifies the website with the transaction ID and redirects the user back to the website; 6) the website receives the notification from the online bank, and adds the corresponding amount on the user's account according to the transaction ID. The bank site only shows the transaction ID on its interface which is usually an unmeaningful string. If the attacker can hijack the step 2 to associate the transaction ID with his account without being noticed, the victim user is likely to finish all steps on the online bank because there is no abnormal visual indication. Once the victim does so, the money is deposited to the attacker's account.

Case-4: Deposit hijacking on JD.com. `JD.com` is a popular E-commerce website in China. In its implementation of the online deposit feature, the second step uses an AJAX request that is not related to any ID-indicator. We have verified that by shadowing `JD.com`'s session cookie "ceshi3.com" at the `AJAX` request, the attacker can hijack the online deposit invisibly, redirecting funds from the victim into the attacker's `jd.com` account.

5.1.5 Account Hijacking in SSO Association

Single Sign On (SSO) is a technique where an Identity Provider (IdP) provides authentication assertions for a logged-in user to relying parties (RP) for them to authenticate the user. SSO usually enables automatic login on the relying party, providing a better user experience and in some cases better security. This is a popular technique deployed by a number of large websites such as Google and Facebook as IdPs, and many other web sites as relying parties. Popular web protocols used for SSO implementation include OpenID [10] and OAuth [14].

Under certain conditions, SSO systems face a threat called *association violation* [31], in which a victim account on an RP is associated with an attacker's account on an IdP, so that the attacker gains control of the victim's account on the RP. This is likely to happen when 1) the victim is logged-in in the IdP as the attacker, 2) the RP has a feature for its users to associate with their accounts on the IdP, 3) the feature is implemented through redirections without further confirmation. The first condition can be mounted by cookie injection, and the websites satisfying the latter two conditions are not hard to find.

Case-5: Account hijacking against Google OAuth and BitBucket. BitBucket, a popular code hosting service, provides account association with Google by OAuth. If a user is already logged in with Google and has authorized BitBucket to access her Google profile through OAuth, the association is accomplished with two forth-and-back redirections with `https://accounts.google.com/o/oauth2/auth` without confirmation except a final message saying *"You've successfully connected your Google account"*.

Our goal is to hijack the Google OAuth URL to invisibly cause an association violation. There are 5 relevant session cookies: "SID/SSID/HSID" with domain scope ".google.com" and path "/", and "LSID/LSOSID" with domain scope ".accounts.google.com" and path "/". This is challenging because the server seemingly has deployed specific defense to counter cookie injection. First, `accounts.google.com` has enabled HSTS with `includeSubDomains`. Second, if we inject cookies with the same names, the server redirects us to a "CookieMismatch" warning page.

We successfully launch the attack on Safari by taking advantage of Safari's quirks. First, we exploit the HSTS implementation bug (Section 4.2) to inject the attacker's five session cookies with domain scope ".accounts.google.com" and path "/o/oauth2/". Recall that Safari ranks cookies by domain specificness then by path specificness, therefore the injected cookies shadows the legitimate ones. Then, we make use of Safari's 8 KB limitation on the cookie header (see Section 4.1) to get around the same name cookie detection. To achieve this, we inject a number of cookies with specific names and domain/path scopes, so that they are ranked between the injected session cookies and the legitimated session cookies. We control the length of these cookies to "overflow" the cookie list so that Safari truncates the legitimated session cookies when issuing requests to the OAuth URL. This allows us to bypassed all restrictions.

5.2 Cookies as References to Session Independent States

Session fixation is a well-known attack in which an attacker holds a session ID, then persuades a victim to authenticate with that ID so that he gains control of the victim's account [21]. Cookie injection can be used to exploit vulnerable websites that use cookies to store session IDs. Standard defenses, *e.g.* regenerating session ID after login, have been widely adopted.

However, we found that, although some websites have implemented defenses for typical session fixation attacks, they still have similar vulnerabilities for cookie injection. The root cause is that they associate important server-side states with long-term cookies. Moreover, they do not bind these states with user sessions.

The attacker can fixate such cookies by cookie injection (*e.g.*, through cookie overwriting) in order to access and manipulate the associated states. Interestingly, most of the vulnerable websites we found vulnerable are E-commerce websites.

Case-6: Shopping cart tracking/manipulation on popular E-commerce websites. We demonstrate this type of issues on 3 popular E-commerce websites: Apple, eBay, and JD.com. These websites allow unregistered visitors to add items in shopping carts. For better user experience, they never expire the shopping carts on the server side, and use long-term cookies on the browser side as references. We have verified that if the attacker fixates the corresponding cookies using cookie injection, he can track or manipulate shopping carts of the unregistered visitors.

We also found similar problems on Amazon, which are much more serious in terms of the real-world consequences, because they compromise security for registered users.[5]

Case-7: Browsing history and purchase tracking/hijacking on Amazon. Amazon's E-commerce websites use two long-term cookies "session-id" and "ubid-main" to associate with a user's browsing history and the ongoing purchase. Surprisingly, these important states are not associated with the user session (Not as its name suggests, "session-id" is not used for user authentication). Once the attacker fixates the two cookies, he can launch various attacks remotely.

The first exploitation is to track and manipulate the user's browsing history. Amazon keeps all previous viewed items in a user's browsing history. Upon fixating the two referencing cookies, the attacker can track what the victim have viewed on Amazon in real-time. He can also inject unwanted items into the browsing history, which affects the recommendation system.

Moreover, from what we observed, we infer that Amazon keeps a session independent data structure for an on-going purchase, which stores the user, the purchased items, the total amount, the delivery address, and other payment information. The structure is likely created by clicking the "proceed to checkout" button, and released after clicking of the "place your order" button. This structure is associated with the same two cookies referencing the browsing history. By fixating the two cookies and consequently gaining access of the data structure, the attacker has various ways to manipulate the victim's purchase remotely. Below we describe two exploitations:

- **Tracking of all purchases.** First, the attacker can track all purchases of the victim. To do so, he first creates an on-going purchase, of which the internal data structure is also shared with the victim. Later, when the victim makes a purchase, the information is updated to the shared data structure, consequently retrieved by the attacker. On Amazon China, the attacker can see all information of the victim's purchase including items, amount, the victim's name, delivery address, and cellphone number. On Amazon U.S., the delivery address and cellphone number are not visible by the attacker.

- **Potential hijacking of purchases.** When detecting an ongoing purchase by the victim, the attacker can change the delivery address so that the purchase is paid by the victim but sent to the attacker. If the victim confirms the hijacked purchase, she cannot even see where the purchase is hijacked to in her order history, because Amazon only shows "Gifting address". The attacker can even manipulate the purchase in such a way that it is paid by the victim, delivered to the attacker, and only recorded in the attacker's order history. The only limitation of the attack is that if the delivery address is new to the payment option, Amazon requires the victim to confirm the card number, however the interface is arguably not alarming. On Amazon China, this limitation does not apply if the victim chooses to pay with a third-party service like Alipay.

5.3 Cookies reflected into HTML

Another common practice is to store auxiliary variables like preferred language or username as cookies, and reflect these cookies into HTML or script snippets. If not implemented carefully, this practice could make websites vulnerable to various attacks in face of cookie injection.

5.3.1 XSS via Cookie Injection

A direct threat is XSS: if reflected cookies are not sanitized sufficiently, the attacker can embed malicious scripts in reflected cookies to launch XSS attacks through cookie injection.

Case-8: XSS via cookie injection on China Construction Bank, Amazon Cloud Drive, eBay and others. We found a number of websites do not validate reflected cookies sufficiently. Using cookie injection, we successfully mounted XSS against China Construction Bank, Amazon Cloud Drive, eBay and several other websites.

Case-9: Insufficient cookie validation on Bank of America. Among the XSS vulnerabilities we found, the one on the Bank of America website is fairly unique. We

[5] However, we note that many E-commerce sites, including Amazon, use mixed content, and thus are also vulnerable to attackers injecting scripts into the insecure domain that remain in the browser cache.

found that one cookie with path "/" on Bank of America's website could be exploited to inject XSS. At first, our naïve exploitation by overwriting the cookie with a XSS payload failed. The limitation was that the website performed a strict validation on the cookie at the login URL. Only if the cookie was absent would the website set a clean value on the cookie from the response of the login URL, then used it in subsequent requests without validation. Our naïve exploitation was prevented by the strict validation at the login URL.

We found a technique to bypass the limitation, so that the XSS payload can be buried into the victim's browser and triggered when next time the victim logs in by injecting multiple cookies. We injected two cookies. The first one had the same 3-tuple identifier as the legitimate one, but with an expired time to ensure the legitimate cookie was discarded and absent at the login URL. The second injected cookie contained the XSS payload and had a different cookie path "/myaccount" that matched with the first URL after login. Although the server set a clean cookie in the response of the login URL, the specific path of the second injected cookie not only avoid being overwritten by the clean cookie, but also shadowed the clean cookie in subsequent requests, triggering a successful XSS attack.

This case implies a possible misconception that performing a complete cookie validation on one "entry point" is sufficient. In fact, because of the asymmetry between cookie read and write operations, every different URL might bring different and unexpected cookie values no matter how server set cookies in previous responses. Developers must treat every request as a new entry point and carefully validate all associated cookies.

5.3.2 BREACH Attacks through Cookie Injection

In 2002, Kelsey observed that when combining encryption with some compression algorithms, the size of compressed data can be used as a side channel, potentially causing plaintext leakage under certain conditions [20]. Rizzo and Duong found a real-world case in 2012, named as CRIME attack, in which an active network attacker initiates encrypted HTTP requests from a victim browser with different URLs as partially-chosen plaintexts, then infer embedded secrets like session cookies by observing the sizes of the compressed and encrypted requests [29]. Rizzo and Duong also mentioned that a similar attack could also be mounted to infer secrets in encrypted HTTP responses. This was explored and demonstrated by Gluck *et al.*, named as the BREACH attack [12].

BREACH requires the attacker to be able to 1) inject a partially-chosen plaintext into the HTTP response of one webpage, and 2) measure the size of the compressed then encrypted response. An active network attacker satisfies the second condition. If a webpage contains a reflected cookie, the attacker can abuse it with cookie injection as the first vector to launch the BREACH attack to infer secrets in this webpage.

Case-10: BREACH attacks on phpMyAdmin and MediaWiki. We found phpMyAdmin, a popular open source web application for remote database management, reflects a cookie for language preference after sanitization in error page if its value is invalid. The BREACH attack using this cookie can reliably infer the CSRF token in the error page, enabling further CSRF attacks. Similarly, MediaWiki reflects a cookie into its login form, also allowing the BREACH attack to infer the CSRF token in the login page.

5.4 Summary

These exploitations show that cookie injection enables undesired and complicated interactions among cookie implementations, web applications, and various known threats. It is clear that our empirical assessment only touches a part of the whole problem space. Nevertheless, we believe these cases demonstrate that the security implications of cookie's lack of integrity are not well and widely understood by the community, and current cookie practices have widespread problems when cookie injection is taken into consideration.

Report and Response. We have reported all vulnerabilities to the affected websites. Some have acknowledged (*e.g.*, Amazon), and some (*e.g.*, Bank of America) have fixed the issues.

6 Possible Defenses

Some existing techniques can help mitigate this threat, including full HSTS, public suffix list, defensive cookie practices, and anomaly detection.

Full HSTS and Public Suffix List. We strongly recommend that websites deploy full HSTS to prevent cookie injection from active network attackers, as this provides complete protection once a site is pinned by a user visit. The community should also make the effort to raise the awareness of cookie injection attacks, and clarify the different levels of security provided by HTTPS, HSTS, and full HSTS. For websites that host shared domains, the best practice is to use separate domains and register them on the public suffix list. Efforts also should be made to increase the awareness of cookie injection from shared domains and the public suffix list.

Defensive Cookie Practices. For websites that cannot enable full HSTS, and have concerns about cookie injection from related domains, defensive cookie practices may mitigate certain cookie injection threats. For example, frequently invalidating session cookies could

reduce the risk of sub-session hijacking. Instead of using cookies, Websites can also use new features in HTML5 like `localStorage` and `sessionStorage` to facilitate browser-side state management, which does not have cookie's integrity deficiencies, although these mechanisms are less convenient for cross-protocol and cross-domain state sharing.

Anomaly Detection. Websites should consider detecting same name cookies in the cookie header, as we discussed in the `accounts.google.com` case. This is reasonable because same name cookies should not be considered a legitimate use according to both the specification and the inconsistent implementations. This detection would protect non-Safari users from attacks using cookie shadowing.

6.1 Proposed Browser Enhancements

We propose several browser-side enhancements to mitigate cookie injection attacks. Our proposals do not require any server-side change, so they would benefit many legacy websites.

6.1.1 Mitigating Active Network Attackers

Currently, Chrome, Firefox and Safari, but not Internet Explorer, have deployed the HSTS support. We believe that if all major browsers could deploy it, websites with full HSTS would be capable of defending against active network attackers in most cases. However, deploying full HSTS needs all subdomains to support HTTPS with valid certificates. There are a number of practical hurdles for websites to satisfy such a strict requirement. For example, Google cannot enable full HSTS, because it is required to support non-HTTPS access for mandatory adult-content filtering at school and some other locations [13]. Kranch and Bonneau also reported the current incapability of Facebook and Twitter to deploy full HSTS [22]. Hence, we believe full HSTS is not likely to be adopted widely in the near future.

To protect a site which cannot deploy full HSTS, a browser must not allow an HTTP connection to replace or shadow secure cookies, effectively adding an HSTS-like pin for any secure cookie. We propose to modify the semantics of the existing cookie store by adding a "do not send" flag and changing the cookie store behavior with the following semantics. We believe these semantic change should provide protections while minimizing the disruption to existing sites:

1. A browser MUST NOT accept a cookie presented in an HTTP response with the `secure` flag set, nor overwrite an unexpired `secure` cookie, except the case in 5.
2. Cookies with the `secure` flag MUST be given higher priority over non-secure cookies.
3. A browser MUST only send the highest priority cookie for any cookie name.
4. In removing cookies due to a too-full cookie store, the browser MUST NOT remove a `secure` cookie when there are non-secure cookies that can be removed.
5. The browser MUST allow an HTTP connection to clear a `secure` cookie by setting an already-expired expiration date, but the browser MUST NOT remove the cookie from the store. Instead, the browser MUST set the "do not send" flag and maintain the original expiration date.
6. The browser MUST NOT send a cookie with the "do not send" flag, nor send any non-secure cookie with the same name.

The first rule prevents an active network attacker from injecting or replacing `secure` cookies. The second and third rules combined prevent an active network attacker from shadowing a secure cookie. The fourth rule prevents an attacker from flooding the cookie store to evict secure cookies. The fifth and sixth rules are subtle but necessary: mixed-content sites might have a "logout" button in HTTP which clears `secure` session cookies. We wish to enable this functionality without allowing attackers to remove and replace a `secure` cookie.

Taken together, our proposals should add HSTS-like pinning to secure cookies within the existing cookie store. If a cookie was set with `secure` flag, an active network attacker can only delete it, which largely mitigates cookie injection attacks [6].

Compatibility. We implemented the first three rules as a Chrome extension[7], and used the extension to manually examined the Alexa top 40 websites. We found one broken case: the signing out operation on `http://www.bing.com/` results in a request-reply with `http://login.live.com/logout.srf` which expires several `secure` session cookies under its SSO IdP domain `live.com`. Allowing HTTP to clear secure cookies should improve compatibility with such signing-out practice.

We also crawled the Alexa top 100,000 domains with both HTTP and HTTPS. In total, 48,039 domains responded with cookies. 152 (0.32%) domains returned `secure` cookies over HTTP; 570 (1.19%) domains responded with cookies that have same name yet different

[6] The non-conforming cookie name behaviors of PHP, ASP, and ASP.NET described in Section 4.1 still expose some possibilities for cookie shadowing. We suggest vendors to fix these incorrect implementations.

[7] `https://github.com/seccookie/ExtSecureCookie`

domains and/or paths. These numbers suggest `secure` cookies over HTTP (incompatible with the first rule) and same name cookies (related to the second and third rules) are rare in real-world websites. We manually examined 10 domains in each case with our extension and did not observe evidence of broken behaviors.

While the results from our compatibility testing are promising, we acknowledge they are preliminary. First, we may have missed subtle incompatibility issues in our manual testing. Second, some incompatibility behaviors may only occur with logged-in sessions and/or specific paths, which our testing may have failed to uncover. We hope our limited experiments will motivate browser vendors to conduct large-scale in-depth compatibility evaluation.

6.1.2 Mitigating Web attackers

A domain can set cookies with a more specific domain scope (*e.g.* host-only) to prevent cookie stealing by XSS from sibling domains. But this currently has no effect on cookie injection since injected cookies with shared domain scopes yet longer paths are effective for cookie shadowing, and longer paths are available in most cases if an adversary is in control of a related domain. Combining the second rule of the above proposals, we suggest:

7. When issuing a request, the browser MUST rank the cookie list by a) presence of the `secure` flag, and b) specificity of the domain scope.

Together with the third rule presented above, this should enable developers to prevent cookies from being overwritten or shadowed by using specific domain scope (together with the `secure` flag when using HTTPS). We have also implemented this policy in the same Chrome extension mentioned above.

7 Related Work

Comparison to Previous Work. We are aware of several research papers that are directly related to cookie's weak SOP and integrity problem [4, 30, 24, 23, 5, 2, 32], and some other papers that are comparable to our work [17, 22]. Among the directly related research, Barth's [2] and Zalewski's work [32] focused on explaining the cause of the cookie integrity problem. Most previous research only briefly touched cookie integrity as a relevant subproblem rather than main topic [4, 30, 24, 23]. Bortz *et al.* 's research is close to ours. Especially, they introduced the notion of a related domain attacker which we use throughout this paper. However, their work is limited to high-level discussion [5]. In summary, previous research has discussed the problem of cookie's lack of integrity, its root cause, and its security implications. However, prior understanding of the subtlety, prevalence, and severity of this problem in the real world is limited. We take a much closer look at the problem space, provide a number of new empirical assessments which we believe will help the community understand the problem more deeply and know the status quo better. Specifically, we conduct a detailed measurement of full HSTS adoption and reveal the threat to CDN customers. Prior to our work, Kranch and Bonneau recently studied full HSTS deployment practice but within a different context [22]. The cookie related problems revealed in our assessment of browser and server libraries are largely unknown, except a few fragmented knowledge from Lundeen's [23] and Lundeen *et al.* 's work [24]. The attack cases we present also supplement previous discussion on potential exploitations in both breadth and depth. Our close-up study also leads us to find promising cookie isolation enhancements that only require browser-side adoption. In contrast, the previous proposed defenses need both browser- and server-side changes [4, 5].

Broadly, our work can be viewed as an in-depth case study of inconsistent access control policies in web. Jackson and Barth's [17] and Singh *et al.* 's work [30] explored this general problem, and each provided various instances. One example illustrated by Jackson and Barth is the ability of JavaScript to read all cookies with matching domain scopes regardless of their paths [17]. This behavior has now been noted explicitly in the current specification [2].

Security Related Cookie Measurement. Zhou and Evans studied the rare deployment of the `HTTPOnly` cookies at the time [33]. They believed that the requirement of both client and server changes played an important hurdle in its adoption. Kranch and Bonneau found many websites deploy HSTS yet do not marked their cookies with the `secure` flag, which are vulnerable to cookie theft in certain conditions [22]. These two measurements were concerned with cookie's confidentiality, while our work looks at the other property, *i.e.* cookie's integrity. Singh *et al.* measured the real-world usages of `secure` cookies (0.07%, 62 out of 89,222 sites) over HTTP and same name cookies (they called duplicate cookies) (5.48%, 4,893 out of 89,222 sites) [30]. Our assessment obtains similar results.

8 Conclusions

Cookies lack integrity. Although long known in community lore, the community has under-appreciated the implications. We have attempted to systematically evaluate the implications of cookie integrity, including evaluating weaknesses and evaluation artifacts in both browser and server libraries, building real-world attacks against ma-

jor sites including Google and Bank of America, including subtle account-hijack attacks and XSS attacks buried in injected cookies, and developing an alternate browser cookie policy that mitigates the threat from network-level attackers. We expect our work to raise the awareness of the problem, and to provide a context for further discussion among researchers, developers and vendors.

Acknowledgements

We would like to thank our shepherd Hovav Shacham, and the anonymous reviewers for their insightful comments. We are grateful to Vern Paxson, Frank Li and David Fifield for valuable discussion, and Jianjun Chen for help of some exploitations. We also thank Chris Evans, Joel Weinberger, Chris Palmer, and Nick Sullivan for valuable feedback. This work is partially supported by the National Natural Science Foundation of China (Grant No. 61472215), Tsinghua National Laboratory for Information Science and Technology (TNList) Academic Exchange Foundation, and the Natinoal Science Foundation (CNS-1213157 and CNS-1237265).

References

[1] Edit This Cookie. http://www.editthiscookie.com/. [accessed Feb-2015].

[2] BARTH, A. HTTP State Management Mechanism. *IETF RFC 6265* (2011).

[3] BARTH, A. The Web Origin Concept. *IETF RFC 6454* (2011).

[4] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th CCS* (2008), ACM, pp. 75–88.

[5] BORTZ, A., BARTH, A., AND CZESKIS, A. Origin Cookies: Session Integrity for Web Applications. *Web 2.0 Security and Privacy (W2SP)* (2011).

[6] CHEN, S., MAO, Z., WANG, Y.-M., AND ZHANG, M. Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments. In *Proceedings of the 30th IEEE S&P (Oakland)* (2009), IEEE, pp. 347–359.

[7] EVANS, C. Cookie Forcing. http://scarybeastsecurity.blogspot.com/2008/11/cookie-forcing.html, 2008. [accessed Feb-2015].

[8] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol–HTTP/1.1. *IETF RFC 2616* (1999).

[9] FIELDING, R., AND RESCHKE, J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. *IETF RFC 7230* (2014).

[10] FOUNDATION, O. OpenID Authentication 2.0 - Final. http://openid.net/specs/openid-authentication-2_0.html. [accessed Feb-2015].

[11] GITHUB. Yummy Cookies across Domains. https://github.com/blog/1466-yummy-cookies-across-domains, 2013. [accessed Feb-2015].

[12] GLUCK, Y., HARRIS, N., AND PRADO, A. BREACH: Reviving the CRIME Attack. http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf, 2013. [accessed Feb-2015].

[13] GOOGLESUPPORT. Block Adult Content at Your School. https://support.google.com/websearch/answer/186669?hl=en. [accessed Feb-2015].

[14] HARDT, D. The OAuth 2.0 Authorization Framework. *IETF RFC 6749* (2012).

[15] HODGES, J., JACKSON, C., AND BARTH, A. Http Strict Transport Security (HSTS). *IETF RFC 6797* (2012).

[16] IEBLOG. Project Spartan and the Windows 10 January Preview Build. http://blogs.msdn.com/b/ie/archive/2015/01/22/project-spartan-and-the-windows-10-january-preview-build.aspx. [accessed Feb-2015].

[17] JACKSON, C., AND BARTH, A. Beware of Finer-Grained Origins. *Proceedings of 2th W2SP* (2008).

[18] JACKSON, C., AND BARTH, A. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *Proceedings of the 17th WWW* (2008), ACM, pp. 525–534.

[19] JOHNSTON, P., AND MOORE, R. Multiple Browser Cookie Injection Vulnerabilities. http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt, 2004. [accessed Feb-2015].

[20] KELSEY, J. Compression and Information Leakage of Plaintext. In *Fast Software Encryption* (2002), Springer, pp. 263–276.

[21] KOLŠEK, M. Session Fixation Vulnerability in Web-based Applications. http://www.acros.si/papers/session_fixation.pdf, 2002. [accessed Feb-2015].

[22] KRANCH, M., AND BONNEAU, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *Proceedings of the 22th NDSS* (2015).

[23] LUNDEEN, R. The Deputies Are still Confused. *Blackhat EU* (2013).

[24] LUNDEEN, R., OU, J., AND RHODES, T. New Ways Im Going to Hack Your Web App. *Blackhat AD* (2011).

[25] MOZZILA. Public Suffix List. https://publicsuffix.org/. [accessed Feb-2015].

[26] NGINX. Module ngx_http_core_module. http://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers. [accessed Jun-2015].

[27] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer networks 31*, 23 (1999), 2435–2463.

[28] PROJECTS, T. C. HTTP Strict Transport Security. http://www.chromium.org/hsts. [accessed Feb-2015].

[29] RIZZO, J., AND DUONG, T. The CRIME Attack. In *EKOparty Security Conference* (2012), vol. 2012.

[30] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the 31th IEEE S&P (Oakland)* (2010), IEEE, pp. 463–478.

[31] WANG, R., ZHOU, Y., CHEN, S., QADEER, S., EVANS, D., AND GUREVICH, Y. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security* (2013), pp. 399–314.

[32] ZALEWSKI, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[33] ZHOU, Y., AND EVANS, D. Why Arent HTTP-only Cookies More Widely Deployed. *Proceedings of 4th W2SP 2* (2010).