Upgrading HTTPS in Mid-Air:
An Empirical Study of Strict Transport Security and Key Pinning

Michael Kranch
Princeton University
mkranch@princeton.edu

Joseph Bonneau
Princeton University
jbonneau@princeton.edu

***Abstract*—We have conducted the first in-depth empirical study of two important new web security features: strict transport security (HSTS) and public-key pinning. Both have been added to the web platform to harden HTTPS, the prevailing standard for secure web browsing. While HSTS is further along, both features still have very limited deployment at a few large websites and a long tail of small, security-conscious sites. We find evidence that many developers do not completely understand these features, with a substantial portion using them in invalid or illogical ways. The majority of sites we observed trying to set an HSTS header did so with basic errors that significantly undermine the security this feature is meant to provide. We also identify several subtle but important new pitfalls in deploying these features in practice. For example, the majority of pinned domains undermined the security benefits by loading non-pinned resources with the ability to hijack the page. A substantial portion of HSTS domains and nearly all pinned domains leaked cookie values, including login cookies, due to the poorly-understood interaction between HTTP cookies and the same-origin policy. Our findings highlight that the web platform, as well as modern web sites, are large and complicated enough to make even conceptually simple security upgrades challenging to deploy in practice.**

I. Introduction

HTTPS [1], which consists of layering HTTP traffic over the TLS/SSL encrypted transport protocols [2] to ensure confidentiality and integrity, is the dominant protocol used to secure web traffic. Though there have been many subtle cryptographic flaws in TLS itself (see [3] for an extensive survey), the most significant problem has been inconsistent and incomplete deployment of HTTPS. Browsers must seamlessly support a mix of HTTP and HTTPS connections, enabling *stripping attacks* [4] whereby network attackers attempt to downgrade a victim's connection to insecure HTTP despite support for HTTPS at both the server and client.

The primary countermeasure to HTTPS stripping is *strict transport security* (HSTS) [5] through which browsers learn that specific domains must only be accessed via HTTPS. This policy may be specified dynamically by sites using an HTTP header or preloaded by browsers for popular domains. While HSTS is conceptually simple, there are subtle interactions with other browser security features, namely the same-origin policy and protections for HTTP cookies. This leads to a number of deployment errors which enable an attacker to steal sensitive data without compromising HTTPS itself.

Beyond HTTPS stripping, there are growing concerns about weaknesses in the certificate authority (CA) system. The public discovery of commercial software to use *compelled certificates* [6], as well as high-profile compromises of several trusted CAs, have spurred interest in defending against a new threat model in which the attacker may obtain a *rogue certificate* for a target domain signed by a trusted CA.

While many protocols have been proposed to maintain security against an attacker with a rogue certificate for a target domain signed by a trusted CA [3], the only defense deployed to date is *public-key pinning* (or just *key pinning*), by which a browser learns to only connect to specific domains over HTTPS if one of a designated set of keys is used. This policy can be used to "pin" a domain to a whitelist of keys of which at least one must appear somewhere in the server's certificate chain. This can be used to pin a domain to specific end-entity keys, certificate authorities, or a mix of both. Key pinning is currently deployed only as a browser-preloaded policy with Chrome and Firefox, although support is planned for dynamically declared header-based pins [7].

While both technologies are still in the early stages, there is significant enough deployment to draw some meaningful lessons about their use in practice. In this work, we report on the first comprehensive survey of HSTS and key pinning. We use OpenWPM [8] to perform realistic crawling (see Section III) of both the list of domains with preloaded security policies in Firefox and Chrome and the top million most highly-visited domains as provided by Alexa [9], examining both sites' static code and dynamically generated traffic.

We catalog several common security bugs observed in practice (see Sections IV–VI). A summary of these errors is provided in Table I. To the best of our knowledge, we provide the first published evidence of these bugs' appearance in the wild. Knowledge of these errors is useful for any administrator seeking to deploy HSTS and/or pinning securely.

In summarizing our findings (Section VIII), we highlight several underlying causes of these errors. In particular, we believe the specification of HSTS and pinning suffers from both insufficient flexibility and a lack of sensible defaults. These lessons are timely given the considerable amount of ongoing research and development of new proposals for upgrading HTTPS security.

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23xxx

TABLE I. SUMMARY OF MAIN VULNERABILITIES FOUND

		Prevalence			
Error	§	%	#	studied domains	Security implications
Preloaded HSTS without dynamic HSTS	IV-E	34.6%	349/1,008	domains with preloaded HSTS	HTTPS stripping possible on old browsers
Erroneous dynamic HSTS configuration	IV-E	59.5%	7,494/12,593	top 1M domains attempting to set HSTS	HTTPS stripping possible
Pinned site with non-pinned active content	V	3.0% 55.6%	8/271 5/9	base domains with preloaded pins non-Google base domains with preloaded pins	data theft with a rogue certificate
Pinned site with non-pinned passive content	V	3.0% 44.4%	8/271 4/9	base domains with preloaded pins non-Google base domains with preloaded pins	page modifications with a rogue certificate
Cookies scoped to non-pinned subdomains	VI-C	1.8% 44.4%	5/271 4/9	base domains with preloaded pins non-Google base domains with preloaded pins	cookie theft with a rogue certificate
Cookies scoped to non-HSTS subdomains	VI-B VI-B	23.8% 47.8%	182/765 2,460/5,099	base domains with preloaded HSTS base domains with dynamic HSTS	cookie theft by active network attacker cookie theft by active network attacker

II. OVERVIEW OF WEB SECURITY TECHNOLOGIES

The core protocols of the World Wide Web, namely HTTP and HTML, were not designed with security in mind. As a result, a series of new technologies have been gradually tacked on to the basic web platform. All of these are to some extent weakened by backwards-compatibility considerations. In this section we provide an overview of relevant web security concepts that we will study in this paper.

A. HTTPS and TLS/SSL

HTTPS is the common name for "HTTP over TLS" [1] which combines normal HTTP traffic with the TLS [2], [10], [11] (Transport Layer Security) protocol instead of basic (insecure) TCP/IP. Most HTTPS implementations will also use the older SSL v3.0 (Secure Sockets Layer) protocol [12] for backwards compatibility reasons although it contains a number of cryptographic weaknesses. TLS and SSL are often used interchangeably to describe secure transport; in this paper we will strictly refer to TLS with the understanding that our analysis applies equally to SSL v3.0.

The goals of TLS are confidentiality against eavesdroppers, integrity against manipulation by an active network adversary, and authenticity by identifying one or both parties with a certificate. The main adversary in TLS is typically called a *man in the middle* (MitM) or *active network attacker*, a malicious proxy who can intercept, modify, block, or redirect all traffic. Formally, this adversary is referred to as a Dolev-Yao attacker model [13]. We will not discuss cryptographic issues with TLS; Clark and van Oorschot provide a thorough survey [3].

1) Certificates and Certification Authorities: The ultimate goal of HTTPS is to bind the communication channel to the legitimate server for a given web domain, and is achieved with the use of server certificates in TLS.[1] Names are bound at the domain level and are sometimes referred to as "hostname," "host," or "fully-qualified domain name". In this paper, we'll always use "domain" to refer to a fully-qualified domain name. We'll use the term "base domain" to refer to the highest-level non-public domain in a fully-qualified domain name, also sometimes referred to as "public suffix plus 1" (PS+1).[2] For example, for `www.example.com` the base domain is `example.com`.

HTTPS clients will check that the "common name" field in the server's presented certificate matches the domain for each HTTP request. The exact rules are somewhat complicated [14] enabling the use of wildcards and the "subject alternative name" extension to allow certificates to match an arbitrary number of domains.

If name matching fails or a certificate is expired, malformed, or signed by an unknown or untrusted certificate authority, HTTPS clients typically show the user a warning. Browser vendors have made the warnings more intrusive over time and click-through rates have declined significantly [15], [16] to below 50% for Firefox. This decrease is generally considered a positive development as the vast majority of HTTPS warning messages represent false positives due to server misconfigurations or expired certificates [17], [18].

B. Strict transport security

Because a large portion of web sites only support insecure HTTP, user agents must support both HTTP and HTTPS. Many domains serve traffic over both HTTP and HTTPS. This enables an active network attacker to attempt to downgrade security to plain HTTP by intercepting redirects from HTTP to HTTPS or rewriting URLs contained in an HTTP page to change the protocol from HTTPS to HTTP. Such an attack is called an *SSL stripping* or *HTTPS stripping* attack. While the threat has long been known [19], it gained increased attention in 2009 when the popular *sslstrip* software package was released to automate the attack [4].

Browsers use graphical indicators to tell the user whether a connection is made over HTTP or HTTPS. Typically, this UI includes showing `https` in the browser's address bar and a padlock icon. However, user studies have indicated that the vast majority of users (upwards of 90%) do not notice if these security indicators are missing and are still willing to transmit sensitive data such as passwords or banking details [20]. Thus, it is insufficient to rely on users to detect if their connection to a normally-secure server has been downgraded to HTTP by an attacker.

To counter the threat of HTTPS stripping, Jackson and Barth proposed "ForceHTTPS" [21] to enable servers to request clients only communicate over HTTPS. Their proposal was ultimately renamed "HTTP strict transport security" and

[1] TLS also supports mutual authentication in which both client and server are identified with a long-term certificate. However, in nearly all use on the web only the server is authenticated and the client authentication is left for higher-level protocols (typically passwords submitted over HTTPS).

[2] Detecting base domains is not as simple as finding the domain immediately below the top-level domain (TLD+1) due to the existence of *public suffixes* of more than one domain, such as `.ca.us` or `.ac.uk`. We use Mozilla's public suffix list (https://publicsuffix.org/) as the canonical list.

standardized in RFC 6797 [5].[3]

1) HSTS security model: HSTS works as a binary (per domain) security policy. Once set, the user agent must refuse to send any traffic to the domain over plain HTTP. Any request which would otherwise be transmitted over HTTP (for example, if the user clicks on a link with the `http` scheme specified) will be upgraded from HTTP to HTTPS.

In addition to upgrading all traffic to HTTPS, the HSTS specification recommends two other changes. First, any TLS error (including certificate errors) should result in a *hard fail* with no opportunity for the user to ignore the error. Second, it is recommended (non-normatively) that browsers disable the loading of insecure resources from an HSTS-enabled page; this policy has since been adopted by Chrome and Firefox for all HTTPS pages even in the absence of HSTS (see Section V).

By default, HSTS is declared for a specific fully-qualified domain name, though there is an optional `includeSubDomains` directive which applies to all subdomains of the domain setting the policy. For example, if `example.com` sets an HSTS policy with `includeSubDomains`, then all traffic to `example.com` as well as `a.example.com` and `b.a.example.com` must be over HTTPS only.[4]

Note that while HSTS requires a valid TLS connection, it places no restrictions on the set of acceptable certificates beyond what the user agent would normally enforce. That is, HSTS simply requires *any valid TLS connection* with a trusted certificate. It is not designed as a defense against insecure HTTPS due to rogue certificates (see Section II-C), only against the absence of HTTPS completely.

2) HSTS headers: The primary means for a server to establish HSTS is by setting the HTTP header [5] `Strict-Transport-Security`. Compliant user agents will apply an HSTS policy to a domain once the header has been observed over an HTTPS connection with no errors. Note that setting the header over plain HTTP has no effect although a number of sites do so anyway (see Section IV-E).

In addition to the optional `includeSubDomains` directive, an HSTS header must specify a `max-age` directive instructing the user agent on how long to cache the HSTS policy. This value is specified in seconds and represents a commitment by the site to support HTTPS for at least that time into future. It is possible to "break" this commitment by serving an HSTS header with `max-age=0` but this directive must be served over HTTPS.

For regularly-visited HSTS domains (at least once per `max-age` period), the policy will be continually updated and prevent HTTP traffic indefinitely. This can be described as a *continuity* policy.[5] A known shortcoming of HSTS is that it can not protect initial connections or connections after extended inactivity or flushing of the browser's HSTS state. HSTS policy caching may also be restricted by browser privacy concerns. For example, policies learned during "private browsing" sessions should be discarded because they contain a record of visited domains.

HSTS is also vulnerable to an attacker capable of manipulating the browser's notion of time so that it thinks a policy has expired, for example by manipulating the NTP protocol [22].

3) HSTS preloads: To address the vulnerability of HTTPS stripping before the user agent has visited a domain and observed an HSTS header, Chrome and Firefox both now ship with a hard-coded list of domains receiving a preloaded HSTS policy.This approach reduces security for preloaded domains to maintaining an authentic, up-to-date browser installation.

Preloaded domains receive an automatic HSTS policy from the browser and may optionally specify `includeSubDomains`.[6] There is no per-domain `max-age` specification, however in Chrome's implementation, the entire preload list has an expiration date if the browser is not regularly updated.

4) HTTPS Everywhere: The EFF's HTTPS Everywhere browser extension[7] (available for Chrome and Firefox) provides similar protection for a much larger list (currently over 5,000 domains). It has been available since 2011. The HTTPS Everywhere extension relies on a large group of volunteers to curate a preload list in a distributed manner. Because it is an optional (though popular) browser extension, HTTPS Everywhere is willing to tolerate occasional over-blocking errors in return for increased coverage compared to the more conservative preload lists. Because HTTPS Everywhere is crowd-sourced, errors are due to the developers and not site operators themselves. Hence we do not study it in this work.

C. Key pinning

HSTS is useful for forcing traffic to utilize HTTPS; however, it has no effect against an attacker able to fraudulently obtain a signed certificate for a victim's domain (often called a *rogue certificate*) and use this certificate in a man-in-the-middle attack. Because every trusted root in the browser can sign for any domain, an attacker will succeed if they are able to obtain a rogue certificate signed by *any* trusted root (of which there are hundreds [23]–[25]). This vulnerability has long been known and security researchers have obtained several rogue certificates by exploiting social engineering and other flaws in the certificate authority's process for validating domain ownership [3].

However, in 2010 it was reported publicly for the first time that commercial software was available for sale to government agencies to utilize rogue certificates to intercept traffic en masse [6]. This report raised the concern of governments using *compelled certificates* obtained by legal procedures or extralegal pressure to perform network eavesdropping attacks.

In addition to the risk of government pressure, a number of high-profile CA compromises have been detected since

[3] Jackson and Barth [21] originally proposed that servers would request HSTS status by setting a special cookie value, but this was ultimately changed to be an HTTP header.

[4] Note that `includeSubDomains` covers subdomains to arbitrary depth unlike wildcard certificates, which only apply to one level of subdomain [14].

[5] HSTS can also be described as a "trust-on-first-use" (TOFU) scheme. We prefer the term *continuity* which does not imply permanent trust after first use.

[6] The syntax between HSTS preloads and HSTS headers is unfortunately incompatible, with `includeSubDomains` specified in the former and `include_subdomains` used in the latter along with other minor details we will omit for clarity.

[7] https://www.eff.org/https-everywhere

2011[8] [26] including security breaches at Comodo and DigiNotar (which has since been removed as a trusted CA from all browsers) and improperly issued subordinate root certificates from TrustWave and TürkTrust. Collectively, these issues have demonstrated that simply requiring HTTPS via HSTS is not sufficient given the risk of rogue certificates.

1) Pinning security model: Key pinning specifies a limited set of public keys which a domain can use in establishing a TLS connection. Specifically, a key pinning policy will specify a list of hashes (typically SHA-1 or SHA-256) each covering the complete Subject Public Key Info field of an X.509 certificate. To satisfy a pinning policy, a TLS connection must use a certificate chain where at least one key appearing in the chain matches at least one entry in the pin set. This enables site operators to pin their server's end-entity public key, the key of the server's preferred root CA, or the key of any intermediate CA. Pinning makes obtaining rogue certificates much more difficult, as the rogue certificate must also match the pinning policy which should greatly reduce the number of CAs which are able to issue a usable rogue certificate.

In fact, the browsers' default policy can be viewed as "pinning" all domains with the set of all trusted root certificate authorities' keys. Explicit pinning policies further reduce this set for specific domains. Much like HSTS, pinning policies apply at the domain level but an optional `includeSubDomains` directive extends this protection to all subdomains.

The risk of misconfigured pinning policies is far greater than accidentally setting HSTS. HSTS can be undone as long as the site operator can present any acceptable certificate, whereas if a site declares a pinning policy and then can't obtain a usable certificate chain satisfying the pins (for example, if it loses the associated private key to a pinned end-entity key), then the domain will effectively be "bricked" until the policy expires. For this reason, pinning policies often require a site to specify at least two pins to mitigate this risk.

2) Pinning preloads: Chrome has deployed preloaded pinning policies since 2011, although only a handful of non-Google domains currently participate. Firefox shipped preloaded pinning in 2014 with policies for a subset of Chrome's pinned domains plus several Mozilla domains. Like with preloaded HSTS, preloaded pinning policies have no individual expiration date but the entire set expires if the browser is not frequently updated. No other browsers have publicly announced plans to implement preloaded pinning.

3) Pinning headers (HPKP): A draft RFC specifies HTTP Public Key Pinning (HPKP) by which sites may declare pinning policies via the `Public-Key-Pins` HTTP header [7]. The syntax is very similar to HSTS, with an optional `includeSubDomains` directive and a mandatory `max-age` directive. Pinning policies will only be accepted when declared via a valid TLS connection which itself satisfies the declared policy.

Unlike HSTS, the HPKP standard adds an additional `Public-Key-Pins-Report-Only` header. When this policy is declared, instead of failing if pins aren't satisfied, the user agent will send a report to a designated address. This is designed as a step towards adoption for domains unsure if they will cause clients to lose access. Additionally, the standard recommends that user agents limit policies to 60 days of duration from the time they are declared, even if a longer `max-age` is specified, as a hedge against attackers attempting to brick a domain by declaring a pinning policy which the genuine owner can't satisfy.

Currently, no browsers support HPKP and the standard remains a draft. Chrome and Firefox have both announced plans to support the standard when it is finalized.

III. Measurement setup

To study how HSTS and pinning are deployed in practice, we made use of an automated web measurement platform. Our goal was to measure web sites by simulating real browsing as accurately as possible while also collecting as much data as possible. To this end we used a version of the Firefox browser modified only for automation and data capture. Our approach combined *static analysis* of downloaded page content with *dynamic analysis* of all connections actually made by the browser on behalf of rendered page content.

OpenWPM: We utilized OpenWPM as the backbone for our testing. OpenWPM is an open-source web-crawling and measurement utility designed to provide a high level of reproducibility [8]. OpenWPM is itself built on top of the well-known Selenium Automated Browser [27] which abstracts away details such as error handling and recovery.

Static Resources: Selenium provides an interface for inspecting the parsed DOM *after* pages have completed loading. We utilized this interface to extract all tags of interest to check for potential mixed content errors that were not triggered by our browsing.

Dynamic Resources: By itself, neither Selenium nor OpenWPM contain an API to instrument the browser as it executes scripts and loads resources on behalf of the page. We utilized the Firefox add-on system to build a custom Firefox extension to instrument and record all resource calls as the page was loaded to capture dynamic resource loading. Our extension implements the `nsIContentPolicy` interface in the Firefox extension API, which is called prior to any resource being loaded. This interface is designed to allow extensions to enforce arbitrary content-loading restrictions. In our case, we allowed all resource loads but recorded the target URL, the page origin URL, as well as the context in which that request was made, for example if it was the result of an image being loaded or an XMLHttpRequest (Ajax).

Sites tested: We conducted three main crawls in our study. The first was a depth-one[9] sputtering of every domain listed in Chrome and Firefox's preloaded HSTS/pinning lists. We tried fetching both the exact domain and the standard `www` subdomain. Some domains were not accessible, as we discuss in Section IV. We did not attempt to find alternate pages for domains without an accessible home page. Manual inspection suggested that all domains which failed to resolve were either

[8] It should be noted that most of these issues were detected due to Chrome's deployment of key pinning. It is possible a large number of CA compromises occurred before pinning was deployed but evaded detection.

[9] By "depth-one", we mean we crawled the original page and followed every link on that page.

content-delivery networks without a home page by design or domains which have ceased operation.

The second crawl was an expanded crawl of the sites with a preloaded pinning policy. For Twitter, Dropbox, Facebook, and Google domains, we performed this crawl while "logged in," that is, using valid session cookies for a real account. The other pinned domains did not implement a login system.

Finally, we performed a HTTP and HTTPS header-only crawl of the exact domain and `www` subdomain of all sites in the Alexa top million [9] list to test for the presence of HSTS or HPKP. We performed this crawl using ZMap [28], a tool designed for efficient probing of a large number of hosts. For the domains attempting to set HSTS or HPKP headers, we also ran our full OpenWPM crawl to extract cookies and test for various additional issues.

IV. Current deployment

In this section we provide an overview of current deployment of HSTS and pinning using crawling and inspection of the preload lists used by Chrome and Firefox. Our statistics are all based on observations from November 2014.

A. Preload implementations

Chrome's preload list is canonically stored[10] in a single JSON file[11] containing both pinning and HSTS policies.There is also a supporting file specifying named keys referenced in pinning policies. Some are specified as complete certificates (although only the keys are germane for pinning) and other are specified as SHA-1 hashes of keys.

Until 2014, the criteria for inclusion in Chrome's list were not formally documented and inclusion required email communication with an individual engineer responsible for maintaining the list [29], [30]. In mid-2014, Google rolled out an automated web form[12] for requesting inclusion into Chrome's preload list. In addition to submitting this form, domains must redirect from HTTP to HTTPS and set an HSTS header with `includeSubDomains`, a `max-age` value of at least 10,886,400 (one year), and a special `preload` directive. As seen in Figure 1, this automated process has led to a significant increase in the numbers of preloaded entries. However, the new requirements have not been retroactively enforced on old entities and new sites can be added at the discretion of Google without meeting these requirements (e.g. Facebook and Yahoo! were recently added but do not set `includeSubDomains`). Additionally, while requests for inclusion can now be submitted automatically, they are still manually reviewed and take months to land in a shipping version of the browser and there still is not an automated method of submitting preloaded pinsets.

During our crawl, we observed 456 entries on the preload list which present the `preload` token, accounting for 1.0% of the list. Since this token was only recently introduced, this number is a clear indication of both the effect of Google's automated entry and the growth of the HSTS list. We also observed 127 additional sites setting the `preload` token in our top million crawl that are not yet in the preloaded list. Several of these sites (32.3%) are presenting invalid headers so were most likely rejected by Google for inclusion in the list; however, the remaining sites do set valid HSTS header and must be awaiting manual confirmation.

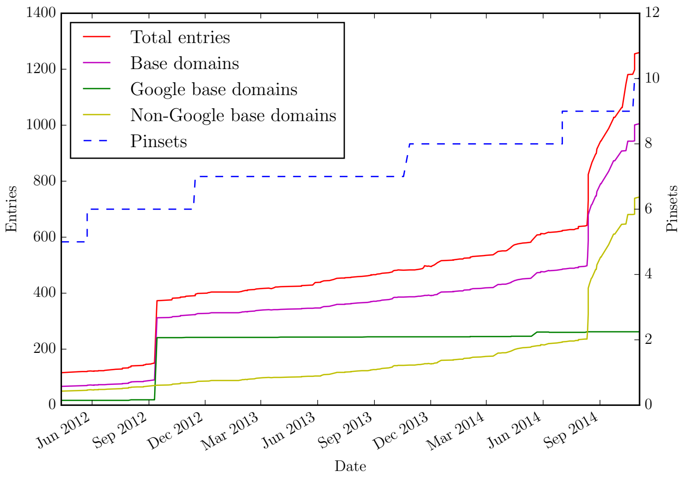

Fig. 1. Growth of preloaded HSTS in Google Chrome.

Mozilla's implementation uses two separate files for preloaded HSTS and pinning policies. The preloaded HSTS file is compiled as C code. Currently, the list is a strict subset of Chrome's list based on those domains in Chrome's list which also set an HSTS header with a `max-age` of at least 18 weeks [31]. The `includeSubDomains` parameter must also be set in the header to be preloaded, so 45 domains are preloaded in Chrome with `includeSubDomains` set but preloaded in Firefox without it. Because Mozilla's list is a strict subset of Chrome's list, we perform all testing for the remainder of the paper on the larger Chrome list.

Firefox also has a preloaded pin list which is implemented as a JSON file with an identical schema to Chrome's. Currently, the only entries are 9 Mozilla-operated domains, 3 test domains, and twitter.com. We perform testing only on Chrome's preloaded pinning lists.

B. Preloaded HSTS

Chrome's preload list, launched in 2010, currently consists of 1258 domains from 1004 unique base domains as summarized in Table II. Roughly a quarter of the current list represents Google-owned properties. Aside from the Google domains, there are a large number of relatively small websites. Figure 2 shows the distribution of preloaded sites' Alexa traffic rankings. The median site rank is about 100,000 and the mean is 1.5M, with 294 base domains (29%) too small to be ranked (and thus not included in our computed mean). Additionally, only 7 of the top 100 and 19 of the top 1000 non-Google sites are included in the preloaded HSTS list, suggesting that uptake is primarily driven by sites' security interest and not by their size. At least 15 sites on the list appear to be individual people's homepages.

[10] This file is not shipped with the browser but is parsed at build-time into a machine-generated C++ file storing the policies

[11] Technically, the file is not valid JSON because it includes C-style comments.

[12] https://hstspreload.appspot.com

TABLE II. Summary of preloaded HSTS and pinning policies

			Google Chrome						Mozilla Firefox	
			total		*Google*		*non-Google*			
HSTS	Pinned	includeSubDomains	Total domains	Base domains	Total domains	Base domains	Total domains	Base domains	Total domains	Base domains
✓	–	–	139	81	0	0	139	81	171	103
✓	–	✓	782	664	0	0	782	664	589	551
–	✓	–	0	0	0	0	0	0	0	0
–	✓	✓	249	243	240	239	9	4	11	5
✓	✓	–	9	7	4	2	5	5	1	1
✓	✓	✓	78	29	56	24	22	5	1	1
all policies			1258	1004	301	262	957	742	773	651

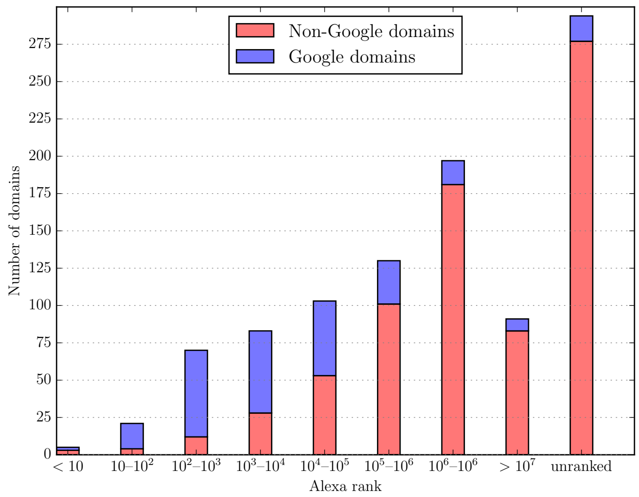

Fig. 2. Histogram of Alexa Site Ranks [9] for domains with a preloaded HSTS policy in Google Chrome.

Even with the relatively small scale and short history of preloaded HSTS, the list is surprisingly stale. Of the 742 non-Google base domains with a preloaded policy, at least 77 domains return a 404 or other site-not-accessible errors. We excluded from this count sites which by manual inspection appeared to be content-delivery networks with no home page by design. These outdated preloaded entries include sites like the infamous liberty.lavabit.com and sunshinepress.org (a long outdated WikiLeaks alias). We also counted 23 sites redirect directly to a distinct plain-HTTP URL, and 3 preloaded HSTS domains redirecting to non-HSTS domain. For example, https://www.evernote.com is the only Evernote entry in the preload list and redirects to https://evernote.com (which does not set a dynamic HSTS header).[13] These may represent stale data or sites may have a legitimate reason to have a changed branding, but the result is that these preloaded policies have little effect in practice. Thus, of the 742 non-Google base domains in the list, 104 (10.3%) represent policies that are no longer needed.

Stale entries were not limited to non-Google domains either: the Google-owned domain urchin.com is still in the list although it was discontinued in May 2012. An additional 4 Google-operated domains with a preloaded HSTS policy now redirect directly to HTTP including ssl.google-analytics.com and dl.google.com.

TABLE III. Summary of preloaded pin sets

Pin set name	# CA pins	# Distinct CAs	# End-entity pins	Total domains	Base domains
cryptoCat	1	1	1	1	1
dropbox	18	4	0	2	1
facebook	3	2	1	16	1
google	2	1	0	300	262
lavabit	0	0	1	1	1
mozilla	21	3	0	6	3
mozilla_services	1	1	0	3	2
tor	2	1	3	5	1
tor2web	1	1	1	1	1
twitterCDN	42	8	1	1	1
twitterCom	21	2	1	6	1

While scalability of the preload list is often discussed in terms of the amount of storage required to ship the list and the cost of checking every browser request for a preloaded policy, these findings highlight that human management of the list is a bigger issue today. All of the stale entries we observed predate the automated process for inclusion. However, it isn't clear if in the future stale entries will be removed or how quickly.

C. Preloaded pinning

A summary of the preloaded pinning policies is shown in Table II. Outside of Google, which pins nearly all of its properties, pinning remains relatively rare with only 36 entries from 9 unique base domains implementing pinning in Firefox or Chrome.

Table III providers further details of the domains that utilize pinning and the size of various pin sets being used. Only 4 for-profit companies (Dropbox, Facebook, Google, and Twitter) are actively utilizing pinning. Of these, Twitter and Dropbox both pin to a large number of certificate authorities rather than end-entity keys. Mozilla takes a similar approach.

By contrast, Facebook's pin set is relatively small with only 3 CA pins (2 being from DigiCert). Google pins to just 2 CAs, although they are intermediate CAs operated by Google itself (Google Internet Authority). Google is rare in controlling a CA as well as largely running its own data centers and content-delivery networks, making it much easier to pin to a smaller set of keys.

Even for domains with a large nominal set of certificate authorities pinned, most in fact come from a small number of organizations as noted in Table II under "Distinct CAs." For example, Twitter's main pin set currently uses 21 certificates but only 2 organizations (DigiCert and Symantec, which owns VeriSign and GeoTrust). The large number of pinned certificates is partially an artifact of historical fragmentation and consolidation in the CA market.

[13] This has since been changed after notification of Evernote.

End-entity pins: Pinning to end-entity keys is rare. Of all pinned domains, we observed only 1 (at tor2web.org) of the 623 keys which matched pinning policies (0.2%) in an end-entity certificate. Lavabit was the only domain specifying a pinning policy consisting of only an end-entity-key, leaving no certificate authority able to undermine its security. Of course, Lavabit suspended operation in August 2013, indicating that the preloaded pin list is also not completely up-to-date.

Thus, while pinning is designed to limit vulnerability to rogue CAs, it is almost exclusively being used to limit trust in CAs and not remove them completely. Furthermore, 75% of active pinsets (not including Lavabit or the test pinset) include a single key (DigiCertEVRoot), meaning that a single CA (DigiCert) can still issue certificates for most pinned domains.

Unused pins: Overall, of the 67 different keys listed in Chrome's `strict_transport_security.certs` file, only 8 are matched when crawling the entire preload list, meaning 88.1% are in the list solely as backup or contingency keys. It is possible some of these keys are used for subdomains that we did not crawl or alternate servers which our crawler did not hit due to load-balancing. However, this large number of unused pins suggests that pinning is challenging for some large websites to deploy as they rely on a huge number of different certificates for different parts of their operation and aren't able to specify a narrow policy.

Pinning without HSTS: Most pinned domains also set HSTS, which is consistent with the belief that pinning is more complicated to deploy and HSTS defends against easier-to-execute attacks. However, 240 Google domains which are pinned do not set HSTS. While Google has not explained this policy, analysis shows 217 of the domains are country-specific variations of the main Google home-page (such as google.sn) as well as google.com itself. For these pages, which implement Google's search engine, supporting non-HTTPS access has been deemed a policy requirement by Google for schools, libraries, and other locations which mandate that adult content is filtered [32]. The remainder are mostly ad network domains (doubleclick.net, googleadservices.com) and content delivery network domains (gstatic.com, googleusercontent.com), but a few security-critical domains are not protected with HSTS, including googleapis.com and android.com. This suggests that, in some cases, enabling pinning may actually be less difficult than HSTS due to the need to support legacy clients which cannot use HTTPS.

Twitter and Tor2web are the only other domains to employ pinning without HSTS for 8 domains. In Twitter's case, these are mostly specific subdomains of twitter.com such as api.twitter.com and mobile.twitter.com, which suggests that support for legacy (non-HTTPS clients) is a significant motivation.

D. Dynamic HSTS

All sites with a preloaded HSTS policy should also be setting HSTS headers. Otherwise no security is provided for HSTS-compliant browsers without a preload list. Still, 257 of the 1,008 preloaded HSTS domains (34.2%) *do not* set HSTS headers. Google domains account for about a sixth of these domains with 39 (65.0% of Google HSTS entries) not setting an HSTS header. It is unclear what Google's reasoning is for not setting HSTS headers for these domains, particularly as Google *does* set an HSTS header on 21 domains including accounts.google.com and plus.google.com. This represents a major security shortcoming as all of these domains are left vulnerable on non-Chrome browsers.

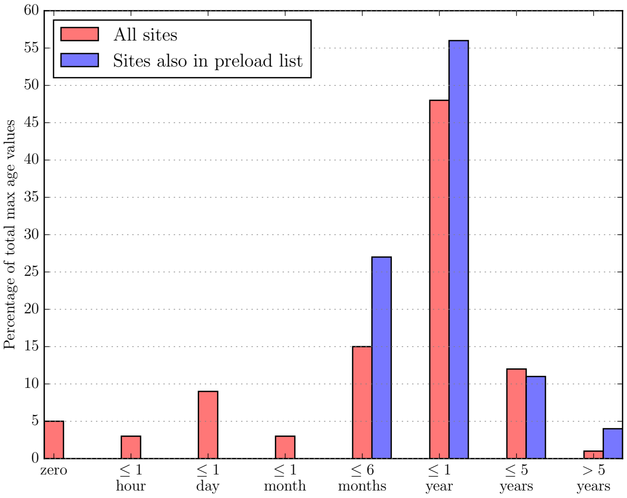

Fig. 3. Histogram of `max-age` values for dynamic HSTS headers.

Additional dynamic HSTS domains: We also crawled all of the Alexa top million websites looking for additional domains setting HSTS headers. We observed only 12,593 sites of the 1,146,249 valid HTTP responses attempt an HSTS header and, of these dynamic HSTS sites, 665 are "opting out" by setting a `max-age=0`.

Even though only a small percentage (1.1%) are setting an HSTS header, 5.8% of the top million sites *should* be using HSTS. We found 66,498 sites using HTTP 30x codes to directly redirect from non-secure HTTP to HTTPS without setting an HTTPS HSTS header, including many financial sites like citibank.com and chase.com, email providers like gmail.com and government sites like healthcare.gov.

Redirecting from HTTP to HTTPS is a strong indication that these sites are intending to only offer their services over a secure connection. The fact that this remains so much more common than setting HSTS indicates developers may not be sufficiently aware of the new technology yet or may be nervous about causing some legacy clients to lose access. It's possible this is an explicit policy choice and the redirects would not have happened if we had crawled with an outdated `user-agent` string. However, by redirecting without setting a header these sites are exposing users to HTTPS stripping every time they connect.

Max-age values: Of the 12,593 attempted HSTS headers we observed, there was a wide range of values for the `max-age` found in practice, as shown in Figure 3. The most common value is 31,536,000 seconds (one year), set by 5,451 domains. This is not surprising given that this value is what is most often used as the example in the HSTS specification [5]. Outside of this standard, the values vary widely with 2,213 headers set at 86,400 seconds (a day) or under including 15 sites that set the ridiculous `max-age` value of one second. Conversely, 1,630 values over a year with the highest value

TABLE IV. DYNAMIC HSTS ERRORS

	Alexa top 1M		Preloaded domains	
	Domains	%	Domains	%
Attempts to set dynamic HSTS	12,593	—	751	—
Doesn't redirect HTTP→HTTPS	5,554	44.1%	23	3.1%
Sets HSTS header only via HTTP	517	4.1%	3	0.4%
Redirects to HTTP domain	774	6.1%	9	3.1%
HSTS Redirects to non-HSTS	74	0.6%	3	0.4%
Malformed HSTS header	322	2.6%	12	1.6%
max-age = 0	665	5.3%	0	0%
0 < max-age <= 1 day	2,213	17.6%	5	0.7%
Sets HSTS securely w/o errors	5,099	40.5%	659	87.7%

found of 100,000,000,000 (over 3,000 years). The max-age values also vary widely amongst the major players with Paypal only setting a header with an age of 4 hours and Twitter, Vine and Etsy all setting max-age over 20 years.

Very short values, such as the 17.6% of domains setting a value of a day or less, arguably undermine the value of HSTS. While they provide some protection against a passive eavesdropper in the case of developer mistakes (such as using the HTTP protocol for an internal link), they are dangerous against an active attacker as many users will be repeatedly making untrusted initial connections if they do not visit the site very frequently.

It is an open question if very large max-age values will cause problems in the future if a small number of clients cache very old policies. Unlike the proposal for HPKP, there is no standard maximum for max-age and user agents really are meant to cache the policy for 80 years if instructed to do so, although most browser instances in practice will not be around for that long.

Finally, we saw 665 sites setting a max-age of 0 including several big domains like LinkedIn and t.co, a Twitter content delivery domain. Yahoo actually redirects from HTTP to HTTPS on every tested Yahoo subdomain but still specifically avoids using HSTS and intentionally sets a max age of 0. This is valid according to the specification [5]. It's intended use is to force clients to forget their cached policy for the domain in case the domain wishes to revert to using HTTP traffic. Of these, 121 redirect from HTTPS to HTTP, clearly using the specification as intended.

E. HSTS errors

Of the 12,593 sites setting dynamic HSTS, 7,494 did so in erroneous ways. A summary of errors we observed is in Table IV. It is striking that overall, of the non-preloaded sites attempting to set HSTS dynamically, nearly 59.5% had a major security problem which undermined the effectiveness of HSTS. The rate of errors was significantly lower among sites with a preloaded HSTS policy.

HSTS sites failing to redirect: The HSTS specification [5] states that HSTS domains *should* redirect HTTP traffic to HTTPS. However, of the 12,593 sites attempting to set dynamic HSTS, 5,554 do not redirect. In addition, 65 preloaded HSTS sites do not redirect from HTTP to HTTPS. In addition to violating the standard this represents a security hole as first-time visitors may never transition to HTTPS and therefore never learn the HSTS policy.

HSTS headers set over HTTP: Another sign of confusion about the technology is sites setting an HSTS header over HTTP. Although the HSTS standard specifies that this has no effect and should not be done [5], we observed 4,094 domains setting an HSTS header on the HTTP version of their sites. For sites otherwise implementing HSTS properly, this is a harmless mistake; however, we found 517 domains set HTTP HSTS headers without specifying a HTTPS header, a strong indication that the sites misunderstood the specification. 1,735 of the total HTTP HSTS domains (including popular tech sites like blockchain.info and getfirebug.com) also failed to redirect to HTTPS, indicating they might not understand that HSTS does not achieve redirection on its own. In addition, 206 preloaded HSTS domains set HSTS headers over HTTP including the pinned site CryptoCat, but these sites all also set valid HTTPS HSTS headers.

These sites are clearly attempting to improve the security of their connection via HSTS but perhaps misunderstanding the nature of the technology. The relatively high proportion of this error compared to sites successfully deploying HSTS (32.5%) is a clear sign of significant developer confusion.

Malformed HSTS: We also found 322 sites setting malformed HSTS headers including notable sites like paypal.com[14] and www.gov.uk. The most common mistake was including more than one value for max-age. For example, www.gov.uk's header includes the key-value pair strict-transport-security: max-age=31536000, max-age=31536000, max-age=31536000;. In all cases of multiple max-ages, the max-age value was the same. We confirmed Chrome and Firefox will tolerate this mistake without harm, but this technically violates the specification. We also found 3 setting simply a value without the required key max-age=. For example, www.pringler.com sets the header strict-transport-security: 2678400). This results in the header being ignored by the browser. We saw a further 3 sites setting a negative value for max-age which causes the header to be ignored.

HSTS redirection to HTTP: We also observed 774 sites which correctly set an HSTS header via HTTPS but then immediately redirected to HTTP (where several then set a meaningless HSTS header). For example, https://www.blockchain.info sets an HSTS header while redirecting to http://blockchain.info. HSTS-compliant browsers handle this error and instead to go to https://blockchain.info which responds successfully, but a non-HSTS-complaint browser would be redirected back to HTTP which is clearly not the site's intention. In addition, 9 preloaded HSTS domains use 30x redirects from the listed preloaded domain to HTTP, 5 of which redirect back to HTTP versions of the same base domain which is still protected by the preload list. The Minnesota state health insurance exchange website, mnsure.org, is both preloaded as HSTS and sets a dynamic HSTS header but still responds with a 301 code (moved permanently) to http://www.mnsure.org/. HSTS-compliant browsers will follow this redirect but then still upgrade the connection to https://www.mnsure.org/.

[14] The error at PayPal was particularly interesting as a PayPal engineer was the lead author of the HSTS standard [5].

F. Dynamic pinning (HPKP) deployment

Dynamic pins, as specified by the HPKP protocol, are not yet implemented by any major browser. Nevertheless, we observed many sites already setting valid HPKP headers indicating they would like to use pinning.

We found 18 domains attempting to set dynamic key pins (of which 7 also had a preloaded HSTS policy). We found a comparable rate of errors with dynamic key pins as with dynamic HSTS with only 12 of 18 (67%) setting pins securely. As with dynamic HSTS, short validity periods were a major issue: 5 set key-pin values of 10 minutes or less (1 at only 60 seconds) and 1 was incorrectly formatted without any `max-age`. It appears most of the domains experimenting with the header are small security-focused sites, with none ranked in the Alexa top 10,000 most popular sites.

In addition, we found two new errors not present with HSTS. Amigogeek.net dynamically pins to a hash that is not found with SHA-1 or SHA-256 in any of the certificates presented by that domain. Segu-info.com.ar mislabels a SHA-1 hash as SHA-256. Both these issues would result in a standards-compliant user agent ignoring these policies.

Most of the key pins we observed were specified as SHA-256 hashes although the standard allows either SHA-1 or SHA-256. This was somewhat surprising as SHA-1 hashes are smaller and therefore more efficient to transmit and all preloaded pins are specified as SHA-1 hashes.

Finally, we observed little use of the HPKP standard's error reporting features. Only one domain (www.mnot.net) set a `Public-key-pins-report-only` header to detect (but not block) pinning errors and only one domain (freenetproject.org) specified the `report-uri` directive to receive error reports.

V. Mixed content

Browsers isolate content using the *same-origin policy*, where the origin is defined as the scheme, host, and port of the content's URL. For example, the contents of a page loaded with the origin `example.com` should not be accessible to JavaScript code loaded by the origin `b.com`. This is a core principle of browser security dating to the early development of Netscape 2.0 [33] and formally specified in the years since [34]. Because HTTP and HTTPS are distinct schemes, the same-origin policy means content delivered over HTTPS is isolated from any insecure HTTP content an attacker injects with the same host and port. Therefore, an attacker cannot simply inject a frame with an origin of `http://example.com` into the browser to attempt to read data from `https://example.com`.

However, subresources such as scripts or stylesheets inherit the origin of the encapsulating document. For example, if `example.com` loads a JavaScript library from `b.com`, the code has an origin of `example.com` regardless of the protocol used to load it and can read user data (such as cookies) or arbitrarily modify the page contents. When an HTTPS page loads resources from an HTTP origin, this is referred to as *mixed content*. Mixed content is considered dangerous as the attacker can modify the resource delivered over HTTP and undermine both the confidentiality and integrity of the HTTPS page, significantly undermining the benefits of deploying HTTPS. For this reason, Internet Explorer has long blocked most forms of mixed content by default, with Chrome in 2011 and Firefox in 2013 following suit [35], although the details vary and there is no standard. Other browsers (such as Safari) allow mixed content with minimal warnings.

Not all mixed content is equally dangerous. While terminology is not standardized, mixed content is broadly divided into *active* content such as scripts, stylesheets, iframes, and Flash objects, which can completely modify the contents of the encapsulating page's DOM or exfiltrate data [36], and *passive* or *display* content such as images, audio, or video which can only modify a limited portion of the rendered page and cannot steal data. All browsers allow passive mixed content by default (usually modifying the graphical HTTPS indicators as a warning). The distinction between active and passive content is not standardized. For example, XMLHttpRequests (Ajax) and WebSockets are considered passive content by Chrome and not blocked but are blocked by Firefox and IE.

A. Pinning and mixed content

Unfortunately, the mixed content problem repeats itself with pinned HTTPS (as it has for HTTPS with Extended Validation certificates [37] and other versions of HTTPS with increased security). If a website served over a pinned HTTPS connection includes active subresources served over traditional (non-pinned) HTTPS, then, just as with traditional mixed content, an attacker capable of manipulating the included resources can hijack the encapsulating page. In the case of non-pinned mixed content, manipulation requires a rogue certificate instead of simply modifying HTTP traffic. It should be noted that this risk is not exactly analogous to traditional mixed content because an attacker's ability to produce a rogue certificate may vary by the target domain whereas the ability to modify HTTP traffic is assumed to be consistent regardless of domain. Still, including non-pinned content substantially undermines the security benefits provided by pinning.

A further potential issue with pinned content is that subresources may be loaded over pinned HTTPS with a different pinned set. This also represents a potential vulnerability, as an attacker may be able to obtain a rogue certificate satisfying the subresource's pin set but not the encapsulating page. Thus, the *effective pin set* of the encapsulating page is the union of the pin sets of all (active) subresources loaded by the page. If any of the subresources are not pinned, security of the page is reduced to the "implicit" set of pins consisting of all trusted root CAs, negating the security benefits of pinning completely.

Empirical results: Overall, from the homepages of 271 total base domains with a pinning policy, we observed a total of 66,537 non-pinned resources being included across 10 domains. Of these, 24,477 resources at 8 domains (dropbox.com, twitter.com, doubleclick.net, crypto.cat, and torproject.org) were active content. As noted above, this effectively negates the security goals of pinning for these domains.

While only 8 of 271 pinned base domains having active mixed-pinning content appears low at first glance, recall that 262 of the pinned domains are operated by Google. Google has been diligent to avoid mixed content and has the advantage of using its own content-delivery and advertising networks.

TABLE V. SELECTED TYPES OF PINNED MIXED CONTENT RESOURCES

	Content type	#
Active	script	15,540
	stylesheet	4,725
	link (rel="stylesheet")	2,470
	xmlhttprequest	1,515
	subdocument	170
	font	49
	total	*24,477*
Passive	image	41,702
	link (rel="shortcut icon")	146
	link (rel="apple-touch-icon")	132
	media	45
	link (rel="image-src")	36
	total	*42,061*

The fact that 5 of the other 9 suffered from fatal active-mixed content problems suggests this will be a serious problem as pinning is incrementally deployed, especially because these sites are on the cutting edge of security awareness.

Sources of mixed content: A summary of the types of resources involved in mixed content errors is provided in Table V. The errors generally arise from including active web analytics or advertising resources. For example, crypto.cat loads 27 scripts on 4 pages (including crypto.cat) from get.clicky.com, a Web analytics site.

Content delivery networks were another major source of errors. At dev.twitter.com we observed 85 loads from various subdomains of the content-delivery network akamai.net. Dropbox was responsible for 921 of the mixed-content loads we observed, including loading scripts, stylesheets (CSS), and fonts (considered active content) from cloudfront.net, another content-delivery network. They load these resources multiple times on essentially every page we visited within the domain.

We also observed interesting errors in "widget" iframes for pinned sites which we happened to observe embedded in other pages in our crawl. For example, Twitter's embeddable gadget twitter.com/settings/facebook/frame loads (3 times) scripts from connect.facebook.net. Similarly, we observed the advertising network DoubleClick loading an assortment of advertising scripts from various locations within an iframe embedded in other sites. While this is meant to be included as an iframe at other sites, the non-pinned scripts it loads could still in some cases steal cookies and read user data. In particular, all of DoubleClick's cookies and many of Twitter's are not marked `httponly` and can therefore be read by malicious scripts.

Impact of subdomains: A large number of these mixed content errors were due to resources loaded from subdomains of pinned domains without `includeSubDomains` set. Of the 9 pinned non-Google base domains, 4 domains had mixed content issues from loading a resource from a non-pinned subdomain of an otherwise pinned domain. Overall, 99.96% of the unpinned active content loads were "self-inflicted" in that they were loaded from the same base domain.

Twitter had perhaps the most issues including loading scripts from syndication.twitter.com. Although they did set a dynamic HSTS Header to protect this resource load from this non-preloaded subdomain, this doesn't fix the fact that the domain isn't pinned. Tor also included content from numerous non-pinned subdomains. Dropbox and CryptoCat both link to their blog and forum subdomain without an HSTS header, and dropbox.com loads images and other passive resources from photo-*.dropbox.com without HSTS being set. The blog.x.com subdomain was the most frequent subdomain with this issue with two of the five domains introducing "self-imposed" mixed content on this subdomain. These findings suggest that confusion over the relationship between subdomains owned by the same entity is a major source of errors and that developers may be forgetting when `includeSubDomains` is in effect.

Some of the problems may also simply come from modern websites' complicated structure making it difficult to remember what is pinned and what isn't. To this end we observed many cases of content included from non-pinned domains owned by the same domain in practice, though not strictly subdomains. For example, Google loads images from *.ggpht.com on many of the major Google domains including play.google.com.

Expanded pin set mixed content: We observed 3,032 references to resources protected by a *different* pin set from 8 domains. As discussed above, this expands the effective pin set to the union of the top-level page and all resources loaded. Of these, 42 were loaded as active content by 2 domains: Twitter and Dropbox. Twitter accounts for over 85% of the expanded pin-set resources, primarily through self-expansion. Since Twitter has two separately listed pin sets, it frequently increases its effective pin set size by loading content from the twitterCDN pin set (e.g. platform.twitter.com and api.twitter.com) on a twitterCom pin set domain. Both Twitter and Dropbox also include a script from ssl.google-analytics.com in multiple places. While this is a lower risk than including unpinned content, these findings support our expectation that mixed content handling will be more complicated for pinned content due to the multiple levels of potential risk..

Plain HTTP resources loaded by pinned domains: We observed a further 30,642 references to resources over plain HTTP from 205 pinned domains. Only one domain, (doubleclick.net), made the mistake (observed 3 times) of including active content over HTTP by including a script from http://bs.serving-sys.com/. Again, this script was only loaded in a doubleclick.net iframe we observed within another page.

These numbers serve as a useful baseline for comparison and suggest that errors due to mixed pinning, particularly active content, are more common than mixed HTTP content. This suggests that this problem is not yet widely understood or appreciated, although it can completely undermine pinning.

B. HSTS Mixed Content

We also briefly consider the existence of mixed content between HSTS-protected HTTPS pages and non-HSTS resources loaded over HTTPS. Unlike the case of pinning, this is not currently a significant security risk because resources referenced via a URL with the `https` scheme must be accessed over HTTPS, even if they are not protected by HSTS.

There is an edge case which is not clearly defined by the specification [5] related to error messages. The HSTS standard requires hard failure with no warning if a connection to an HSTS domain has a certificate error but doesn't specify if warnings can be shown for non-HSTS resources loaded by the page. This is likely a moot point, as modern browsers now typically block active content which would produce a certificate error even from non-HSTS pages.

Still, we found references to non-HSTS resources from HSTS pages were widespread, with 171,533 references from 349 base domains, of which 87,465 from 311 domains were active content. As with the pinned mixed content errors, the vast majority were "self-inflicted" in that they were resources loaded from a common base domain, accounting for 84.73% of all mixed content and 71.96% of the active mixed content. Resources from explicit subdomains were again a major source of mixed policy, with 20,913 references from 115 base domains, of which 10,577 were active content.

VI. Cookie theft

A long-standing problem with the web has been the inconsistency between the same-origin policy defined for most web content and the one defined for cookies [38]–[40]. Per the original cookie specification [38], cookies are isolated only by host and not by port or scheme. This means cookies set by a domain via HTTPS will be submitted back to the same domain over HTTP [41]. Because cookies often contain sensitive information, particularly session identifiers which serve as login credentials, this poses a major security problem. Even if a domain `secure.com` only serves content over HTTPS, an active attacker may inject script into any page in the browser triggering an HTTP request to `http://secure.com/non-existent` and the outbound request will contain all of the users cookie's for the domain.

A. *`Secure` cookies*

To address this problem, the `secure` attribute for cookies was added in 2000 by RFC 2965 [39], the first update to the cookie specification. This attribute specifies that cookies should only be sent over a "secure" connection. While this was left undefined in the formal specification, all implementations have interpreted this to limit the cookie to being sent over HTTPS [40]. A persistent issue with the `secure` attribute is that it protects read access but not write access. HTTP pages are able to overwrite (or "clobber") cookies even if they were originally marked `secure`.[15]

B. *Interaction of `secure` cookies and HSTS*

At first glance, it might appear that HSTS obviates the `secure` cookie attribute, because if a browser learns of an HSTS policy and will refuse to connect to a domain over plain HTTP at all it won't be unable to leak a secure cookie over HTTPS. Unfortunately, there the different treatment of subdomains which means that cookies can still be leaked.

Cookies may include a `domain` attribute which specifies which domains the cookie should be transmitted to. By default, this includes all subdomains of the specified domain, unlike HSTS which does not apply to subdomains by default. Even more confusingly, the only way to limit a cookie to a single specific domain is to not specify a `domain` parameter at all, in which case the cookie should be limited to exactly the domain of the page that set it. However, Internet Explorer violates the standard [39] in this case and scopes the cookie to all subdomains anyway [41].

TABLE VI. Vulnerable cookies from HSTS domains

Condition	Preloaded HSTS domains #	Preloaded HSTS domains %	Dynamic HSTS domains #	Dynamic HSTS domains %
Domains with HSTS hole	230/765	30.1%	3,637/5,099	70.7%
Domains with vulnerable cookies	182/765	23.8%	2,460/5,099	47.8%
Cookies not marked `secure`	782/823	95.0%	10,174/10,398	97.8%

Thus, a well-intentioned website may expose cookies by setting HSTS but not the `secure` attribute if the HSTS policy does not specify `includeSubDomains` (which is the default) and the cookie is scoped to be accessible to subdomains (which occurs whenever any `domain` attribute is set). For example, suppose `example.com`, a domain which successfully sets HSTS without `includeSubDomains`, sets a cookie `session_id=`x with `domain=example.com` but does not set `secure`. This cookie will now be transmitted over HTTP to any subdomain of `example.com`. The browser won't connect over HTTP to `example.com` due to the HSTS policy, but will connect to `http://nonexistent.example.com` and leak the cookie value x over plain HTTP.

An active attacker can inject a reference to `http://nonexistent.example.com` into any page in the browser, making this cookie effectively accessible to any network attacker despite the domain's efforts to enforce security via HSTS. Thus, we consider this to be a bug as it very likely undermines the security policy the domain administrator is hoping to enforce. HSTS does not serve as an effective replacement for `secure` cookies for this reason and it is advisable that HSTS sites generally mark all cookies as `secure` unless they are specifically needed by an HTTP subdomain.

Empirical results: This vulnerability requires three conditions: an HSTS domain with a non-HSTS subdomain (a "hole"), cookies scoped to that subdomain, and those cookies to not be marked with `secure`. Table VI summarizes the number of domains vulnerable to the attack, broken down by these three conditions.

This issue was present on several important domains like Paypal, Lastpass, and USAA, and the cookies included numerous tracking and analytics cookies, user attributes cookies like county code and language, and unique session identification cookies like "guest_id," "VISITORID", and "EndUserId". Stealing these cookies can be a violation of user's privacy and may be used to obtain a unique identifier for users browsing over HTTPS. Encouragingly, however, all authentication cookies we were able to identify for these sites were marked as `secure` and hence could not be leaked over HTTP. This suggests that the `secure` attribute is relatively well-understood by web developers.

C. *Interaction of cookies and pinning*

A similar issue exists for pinned domains whereby a cookie may leak to unprotected subdomains. For example, if `example.com`, a pinned domain without `includeSubDomains`, sets a cookie `session_id=`x with `domain=example.com`, the cookie will be transmitted over unpinned HTTPS to any subdomain of `example.com`. Note that even setting the `secure` flag doesn't help here—this will

[15] In fact, HTTP connections can set cookies and mark them as `secure`, in which case they won't be able to read them back over HTTP.

TABLE VII. LEAKABLE PINNED COOKIES

Domain	Domain Hole	Insecure Cookies	Total Cookies
crypto.cat	*.crypto.cat	3	3
dropbox.com	*.dropbox.com	3	8
facebook.com	*.facebook.com	17	21
twitter.com	*.twitter.com	35	38
www.gmail.com	*.www.gmail.com	5	5
total		63	75

only require the cookie to be sent over HTTPS but an attacker able to compromise HTTPS with a rogue certificate will still be able to observe the value of the cookie.

Because there is no equivalent attribute to `secure` which would require a cookie to be sent over a pinned connection, there is currently no good fix for this problem. The only way to securely set cookies for a pinned domain is either to limit them to a specific domain (by not setting a `domain` parameter) or to specify `includeSubDomains` for the pinning policy.

Empirical results: We checked for cookies vulnerable to theft over non-pinned HTTPS from all pinned domains in the Chrome preload list. We observed 75 cookies on 5 pinned domains which are accessible by non-pinned subdomains, as summarized in Table VII. As mentioned above, there is no equivalent of the `secure` attribute to limit cookies to transmission over a pinned connection, meaning all of these cookies are vulnerable.

Interestingly, the majority of these cookies are also in fact vulnerable to theft over plain HTTP as 63 of these cookies (84.0%) did not set the `secure` attribute.[16] This suggests that even if an attribute existed to limit cookies to a pinned connection, the semantics of this problem are complex enough that developers may not always deploy it.

Unlike our results for HSTS domains, we crawled 4 of the pinned sites with login cookies and we did observe several authentication cookies vulnerable to theft. Notably, authentication cookies[17] for both Twitter (with it's critical `auth_token` cookie) and Facebook (with `c_user` and `xs`) are both vulnerable. Both are scoped to .twitter.com and .facebook.com, respectively, meaning they are visible to all subdomains even though neither Twitter nor Facebook set `includeSubDomains` for their base domain. Thus an attacker can steal valid authentication cookies for either website without triggering the pinning policy.

We responsibly disclosed this vulnerability to both sites. Unfortunately, in both cases it is considered unfixable at the moment as neither site is capable of setting `includeSubDomains` for their preloaded HSTS policy. A fix has been proposed which will allow these sites to specify `includeSubDomainsForPinning` in the preload file.

Dropbox, by contrast, sets pins for dropbox.com without `includeSubDomains` but scoped its login cookies to .www.dropbox.com, for which `includeSubDomains` is set, preventing the cookies from being vulnerable.

Google's case is considerably more complex but instructive. While the majority of Google's pinning entries set `includeSubDomains` including google.com and thus would appear to avoid this error, until August 2014 play.google.com did not set `includeSubDomains`.[18] For subdomains *.play.google.com, the play.google.com this entry overrode the less specific google.com entry as per RFC 6797 [5]. As a result, any subdomain of play.google.com like evil.play.google.com was not be bound by the Google pin set and an adversary with a rogue certificate for one of these domains would have access to all of cookies scoped for *.google.com there. However, Google limits its "master" authentication cookies' scope to accounts.google.com, which cannot be accessed by *.play.google.com, but assigns per-subdomain authentication cookies as needed. Thus this vulnerability was limited to login cookies giving access to play.google.com only. Google was aware of this vulnerability when we initially disclosed it and has since fixed it by extending `includeSubDomains` to cover play.google.com.

Recommendation to browsers: As a result of our findings with pinned cookies, we recommend that browser vendors extend the semantics of the `secure` attribute for cookies as follows: if a cookie is set by a domain with a pinning policy and marked `secure`, the cookie should only be transmitted over HTTPS connections which satisfy the pinning policy of the domain setting the cookie. This is a relatively simple fix which would close the security holes we found without introducing any new syntax. This is also a reasonable interpretation of the original specification for `secure`, which never limited the syntax to mean simply HTTPS. Given that a large number of developers have successfully learned to mark important cookies as `secure`, it makes to extend this in a natural way as pinning and other HTTPS upgrades are deployed.

VII. RELATED WORK

A. Empirical web security studies

Our work fits nicely into a long and fruitful line of measurement studies of web security, covering a wide variety web security topics such as authentication cookies [43], `http-only` cookies [42], password implementations [44], third-party script inclusions [45], [46], third-party trackers [47], [48], Flash cross-domain policies [49], and OpenID implementations [50], [51]. A classic problem is detecting cross-site scripting vulnerabilities which is practically its own sub-field of research [52]–[55]. A common model for this research is exploration and analysis of an emerging threat on the web, followed by measurement and crawling to detect its prevalence and character. A desirable research outcome is for automated detection to be built-in to web application security scanners [56]–[58].

Ultimately, browsers themselves have come to absorb some of this logic. For example, Chrome and Firefox both now warn developers about basic mixed-content errors in their built-in developer consoles. Several issues identified in our research are solid candidates for inclusion in future automated tools: cookies vulnerable to theft, pinning mixed content, and some types of erroneous or short-lived HSTS headers.

[16] Note that because the Chrome preload file requires only one line per domain for both pinning and HSTS policies, every domain with a non-`includeSubDomains` pinning policy also has a non-`includeSubDomains` HSTS policy (if any HSTS policy at all).

[17] Identifying exactly which set of cookies is sufficient to hijack a user's session can be a difficult problem [42] but we confirmed manually for Twitter and Facebook that the vulnerable cookies were sufficient to log in.

[18] Chrome's preload list previously included the comment "play.google.com doesn't have include_subdomains because of crbug.com/327834;" however, this link was never valid and it isn't clear what the original bug was.

B. Empirical studies of HTTPS and TLS

A significant amount of research has also focused specifically on empirical errors with HTTPS and TLS, of which Clark and van Oorschot provide the definitive survey [3]. Important studies of cryptographic vulnerabilities have included studies of key revocation after the Debian randomness bug [59], studies of factorable RSA keys due to shared prime factors [60], [61], studies of elliptic curve deployment errors in TLS [62], forged TLS certificates in the wild [18] and multiple studies of key sizes and cipher suites used in practice [23], [63], [64]. Our work is largely distinct from these in that we focus on two new aspects of HTTPS (pinning and strict transport security) which are vulnerabilities at the HTTPS (application) level rather than the TLS (cryptographic) level.

Perhaps the most similar study to ours was by Chen et al. [65] which focused on mixed-content and stripping vulnerabilities. This study was performed prior to the advent of HSTS, pinning, and mixed content blocking. Hence, that work can be viewed as a first-generation study compared to our second-generation study based on newly introduced security measures (although no doubt many of the original vulnerabilities are still present on the web today).

Other empirical studies of the TLS ecosystem have focused on certificate validation libraries [66], non-browser TLS libraries [67], the interaction of HTTPS with content-delivery networks [68], and TLS implementations in Android apps [69], [70]. Again, these efforts all found widespread weaknesses. A common theme is developers not correctly understanding the underlying technology and using them in an insecure manner.

C. Other proposals for improving HTTPS

We briefly overview several noteworthy proposals for further improving HTTPS. These proposals are mainly aimed at limiting the risk of rogue certificates and hence could complement or supplant key pinning. We exclude other web security proposals such as Content Security Policy (CSP) [71].

1) DANE: DNS-based Authentication of Named Entities (DANE) [72] is a proposal for including the equivalent of public key pins in DNS records, relying on DNSSEC [73] for security. This has the advantage of avoiding the scalability concerns of preloaded pins and potentially being easier[19] and more efficient[20] to configure than pins set in headers. Unfortunately, DANE adoption is delayed pending widespread support for DNSSEC, which common browsers do not currently implement.

DANE does not currently contain support for declaring policies applicable to all subdomains. Based on our study, we would strongly advise such support be added (and possibly turned on by default) to avoid the type of mixed content and cookie vulnerabilities we observed with pinning today.

2) Out-of-chain key pinning: An alternative to pinning to keys within a cite's certificate chain is to specify a separate self-managed public key which must sign all end-entity public keys, in addition to requiring a certificate chain leading to a trusted CA. This avoids fully trusting any external CA while offering more flexibility than pinning to an enumerated set of end-entity public keys. Conceptually, it is similar to pinning to a domain-owned key in a CA-signed, domain-bound intermediate certificate.[21] TACK [74] proposes distributing out-of-chain public keys using continuity,[22] while Sovereign Keys [75] proposes using a public append-only log.

TACK explicitly does not contain support for extending policies to subdomains, instead recommending that implementers omit the domain parameter to limit cookies to one domain and/or set the `secure` flag to avoid cookie theft. Of course, the `secure` flag is inadequate for defending against rogue certificates given that one cannot set a TACK policy for non-existent subdomains. Sovereign Keys, by contrast, does support wildcards to extend support to subdomains.

3) Public logging: Due to the risk of improperly configured pinning policies causing websites to be inaccessible, some proposals aim simply to require that all valid certificates be publicly logged to ensure rogue certificates are detected after the fact. Certificate Transparency [76] (CT) is the most prominent of these efforts, recording all valid end-entity certificates in a publicly verifiable, append-only log. As currently proposed, clients will begin to reject certificates lacking proof of inclusion in the log after universal adoption[23] by all public CAs.[24] A somewhat-related proposal is Accountable Key Infrastructure [77].

As proposed, Certificate Transparency would avoid most of the subtle issues identified in this work in that the burden on web developers is extremely low (the only requirement is to start using a CT-logged certificate prior to some future deadline). Given the number of errors we observed in our study, this seems like a major design advantage. However, if CT is not able to be adopted via the proposed “flag day” strategy, it may be necessary to distribute policies to browsers specifying which domains require CT protection. This problem would be largely similar to distributing HSTS today. Thus, the lessons from our paper would apply to designing such a mechanism, which most likely would be implemented as an extra directive in HSTS itself.

VIII. Concluding discussion

HSTS is still in the early stages of adoption and pinning is in the very early stages of adoption. Nevertheless, both technologies have already greatly enhanced security for a number of important websites. It is a tribute to pinning that it has been responsible for the detection of most of the known CA compromises since it was deployed in 2010. Still our research shows that a large number of misconfiguration errors are undermining the potential security in many cases with its early deployment.

[19] It is an open question whether web developers are more comfortable configuring HTTP headers or DNS TXT records.

[20] Unlike for HSTS which operates with relatively small headers, there is significant concern about the efficiency of setting pins in headers which may be hundreds of bytes long for complicated pin sets as seen in our study.

[21] Domain-bound intermediates are possible using X.509's `nameConstraints` extension. However, this extension is not universally supported and non-supporting clients will reject such certificates.

[22] Continuity in TACK is distinct from HSTS/HPKP in that clients only retain a TACK policy for as long into the future as the policy has been consistently observed in the past, subject to a maximum of 30 days.

[23] Even after universal adoption, clients must wait until all extant legacy certificates have expired to require CT proofs.

[24] Private CAs, such as those used within an enterprise, are excluded.

Some of these issues can be attributed to developer unfamiliarity with the new tools as previous studies have shown that developers traditionally make critical mistakes in the early implementation of new techniques. It is worth emphasizing, however, that many of the errors we observed were made by large websites that are, in many ways, at the forefront of web security, the very developers who should be in the best position to understand these new tools. While familiarity may increase with time, less security-aware developers will begin implementing these techniques which may increase the number of errors.

Our work should serve as a reminder that simplicity for developers is a critical aspect of any web security technology. Considering the root causes of some of the errors we found, better defaults might have helped. For example, we would advocate for a default value of perhaps 30 days for HSTS policies set without an explicit max-age. Forcing all developers to choose this value has probably led to unwise choices on both ends of the spectrum in addition to malformed headers. Setting sensible defaults for pinning is far more challenging—there is no clear way to choose a default "backup pin" besides the values currently being used. This issue suggests that pinning may never be a simple "on switch" for developers unless TLS certificate management can be abstracted away completely.

Another important takeaway from our work is that many developers appear to not fully understand the same-origin policy and the relation of subdomains to one another. This is particularly true with cookies. For both HSTS and pinning, having policies apply to subdomains by default, with an option to disable this or turn it off for specific subdomains, would be a safer design. This recommendation is already reflected in Chrome's new policy for preloaded inclusion which requires sites to set `includeSubDomains`. Extending cookies' `secure` attribute to require pinning (where applicable) as well as HTTPS, as discussed in Section VI-C, also appears to be a simple step towards making the technology match developer expectations more closely.

Finally, we advocate for streamlining HTTPS security features to make configuration as simple as possible. HSTS and pinning will almost certainly not be the last HTTPS enhancements ever added. Already there are two distinct syntaxes being standardized to set them in HTTP headers and browser preloads. It would be beneficial to combine dynamic HSTS and pinning declarations into a more flexible and extensible syntax that developers can declare once, preferably with sensible defaults, rather than expect developers to learn new syntax and subtleties as each new patch is applied.

Acknowledgments

We would like to thank Steven Englehardt, Christian Eubank, and Peter Zimmerman for assistance using OpenWPM. We thank David Adrian, Zakir Durmeric and Alex Halderman for assistance obtaining header data using ZMap. We thank Arvind Narayanan, Jennifer Rexford, Harry Kalodner, and Laura Roberts for feedback on our write-up and presenting our results. We thank Eric Lawrence, Adam Langley, Jeffrey Walton, Tanvi Vyas, and Devdatta Akhawe for volunteering helpful technical fixes to earlier drafts.

References

[1] E. Rescorla, "HTTP over TLS," RFC 2818, Internet Engineering Task Force, 2000.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Internet Engineering Task Force, 2008.

[3] J. Clark and P. C. van Oorschot, "SSL and HTTPS: Revisiting past challenges and evaluating certicate trust model enhancements," *IEEE Symposium on Security and Privacy*, 2013.

[4] M. Marlinspike, "New Tricks For Defeating SSL In Practice," in *Black Hat DC*, 2009.

[5] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," RFC 6797, Internet Engineering Task Force, 2012.

[6] C. Soghoian and S. Stamm, "Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL," *Financial Cryptography and Data Security*, 2012.

[7] C. Evans, C. Palmer, and R. Sleevi, "Internet-Draft: Public Key Pinning Extension for HTTP," 2012.

[8] S. Englehardt, C. Eubank, P. Zimmerman, D. Reisman, and A. Narayanan, "Web Privacy Measurement: Scientific principles, engineering platform, and new results," 2014.

[9] "Alexa: The Web Information Company," http://www.alexa.com, 2014.

[10] T. Dierks and c. Allen, "The Transport Layer Security (TLS) Protocol Version 1.0," RFC 2246, Internet Engineering Task Force, 1999.

[11] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, Internet Engineering Task Force, 2006.

[12] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101, Internet Engineering Task Force, May 2011.

[13] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, 1983.

[14] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," RFC 2459, Internet Engineering Task Force, 1999.

[15] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," *USENIX Security Symposium*, 2009.

[16] D. Akhawe and A. P. Felt, "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness," *USENIX Security Symposium*, 2013.

[17] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: Understanding TLS errors on the web," *22nd International Conference on World Wide Web (WWW)*, 2013.

[18] L.-S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing Forged SSL Certificates in the Wild," *IEEE Symposium on Security and Privacy*, 2014.

[19] A. Ornaghi and M. Valleri, "Man in the middle attack demos," *Blackhat Security*, 2003.

[20] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," *IEEE Symposium on Security and Privacy*, 2007.

[21] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," *Web 2.0 Security and Privacy*, 2008.

[22] Jose Selvi, "Bypassing HTTP Strict Transport Security," Black Hat Europe, 2014.

[23] I. Ristic, "Internet SSL Survey 2010," *Black Hat USA*, 2010.

[24] P. Eckersley and J. Burns, "The (decentralized) SSL observatory (Invited Talk)," *20th USENIX Security Symposium*, 2011.

[25] J. Kasten, E. Wustrow, and J. A. Halderman, "Cage: Taming certificate authorities by inferring restricted scopes," *Financial Cryptography and Data Security*, 2013.

[26] A. Niemann and J. Brendel, "A Survey on CA Compromises," 2013.

[27] J. Huggins and P. e. a. Hammant, "Selenium browser automation framework," *http://code.google.com/p/selenium*, 2014.

[28] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications." *USENIX Security Symposium*, 2013.

[29] A. Langley, "Strict Transport Security," Imperial Violet (blog), January 2010.

[30] ——, "Public Key Pinning," Imperial Violet (blog), May 2011.

[31] D. Keeler, "Preloading HSTS," Mozilla Security blog, November 2012.

[32] Google Support, "Block adult content at your school with SafeSearch," retrieved 2014.

[33] J. Ruderman, "The same origin policy," http://www.mozilla.org/projects/security/components/same-origin.html, 2001.

[34] A. Barth, "The Web Origin Concept," RFC 6454, Internet Engineering Task Force, 2011.

[35] I. Ristic, "HTTPS Mixed Content: Still the Easiest Way to Break SSL," Qualys Security Labs Blog, 2014.

[36] T. Vyas, "Mixed Content Blocking Enabled in Firefox 23!" Mozilla Security Engineering—Tanvi's Blog, 2013.

[37] M. Zusman and A. Sotirov, "Sub-prime PKI: Attacking extended validation SSL," *Black Hat Security Briefings*, 2009.

[38] D. Kristol, "HTTP State Management Mechanism," RFC rfc2109, Internet Engineering Task Force, 1997.

[39] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC 2965, Internet Engineering Task Force, 2000.

[40] A. Barth, "HTTP State Management Mechanism," RFC 6265, Internet Engineering Task Force, 2011.

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

[42] Y. Zhou and D. Evans, "Why Aren't HTTP-only Cookies More Widely Deployed?" *Web 2.0 Security and Privacy*, 2010.

[43] K. Fu, E. Sit, K. Smith, and N. Feamster, "Dos and don'ts of client authentication on the web," in *USENIX Security*. Berkeley, CA, USA: USENIX Association, 2001.

[44] J. Bonneau and S. Preibusch, "The password thicket: technical and market failures in human authentication on the web," *Workshop on the Economics of Information Security (WEIS)*, 2010.

[45] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," *18th International Conference on World Wide Web (WWW)*, 2009.

[46] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[47] J. Mayer and J. Mitchell, "Third-party web tracking: Policy and technology," in *IEEE Symposium on Security and Privacy*, 2012.

[48] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third–party tracking on the web," *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[49] A. Venkataraman, "Analyzing the Flash crossdomain policies," Master's thesis, 2012.

[50] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[51] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services," *IEEE Symposium on Security and Privacy*, 2012.

[52] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Int. Telecommunications Energy Conference (INTELEC)*, 2004.

[53] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," *IEEE Symposium on Security and Privacy*, 2006.

[54] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," *30th International Conference on Software Engineering (ICSE)*, 2008.

[55] A. E. Nunan, E. Souto, E. M. dos Santos, and E. Feitosa, "Automatic classification of cross-site scripting in web pages using document-based and URL-based features," in *2012 IEEE Symposium on Computers and Communications (ISCC)*, 2012.

[56] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," *15th International Conference on World Wide Web (WWW)*, 2006.

[57] M. Curphey and R. Arawo, "Web application security assessment tools," *IEEE Symposium on Security & Privacy*, 2006.

[58] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2009.

[59] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 debian openssl vulnerability," *ACM SIGCOMM Internet Measurement Conference*, 2009.

[60] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, Whit is right," *IACR Cryptology ePrint Archive*, 2012.

[61] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," *USENIX Security Symposium*, 2012.

[62] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, "Elliptic curve cryptography in practice." *IACR Cryptology ePrint Archive*, 2013.

[63] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements," *ACM SIGCOMM Internet Measurement Conference*, 2011.

[64] B. Amann, M. Vallentin, S. Hall, and R. Sommer, "Revisiting SSL: A large-scale study of the internets most trusted protocol," Technical report, ICSI, Tech. Rep., 2012.

[65] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang, "Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments," *IEEE Symposium on Security and Privacy*, 2009.

[66] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," *IEEE Symposium on Security and Privacy*, 2014.

[67] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," *ACM Conference on Computer and Communications Security*, 2012.

[68] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, "When https meets cdn: A case of authentication in delegated service," *IEEE Symposium on Security and Privacy*, 2014.

[69] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[70] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[71] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *19th International Conference on World Wide Web (WWW)*, 2010.

[72] P. Hoffman and J. Schlyter, "RFC 6698: The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA," 2012.

[73] G. Ateniese and S. Mangard, "A new approach to DNS security (DNSSEC)," *ACM Conference on Computer and Communications Security (CCS)*, 2001.

[74] M. Marlinspike and T. Perrin, "Internet-Draft: Trust Assertions for Certificate Keys," 2012.

[75] P. Eckersley, "Internet-Draft: Sovereign Key Cryptography for Internet Domains," 2012.

[76] B. Laurie, A. Langley, and E. Käsper, "Internet-Draft: Certificate Transparency," 2013.

[77] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure," *22nd International Conference on World Wide Web (WWW)*, 2013.