S-links: Why distributed security policy requires secure introduction

Joseph Bonneau
Google Inc.
jcb@google.com

***Abstract*—In this paper we argue that secure introduction via hyperlinks will be essential for distributing security policies on the web. The "strict transport security" policy, which makes HTTPS mandatory for a given domain, can already be expressed by links with an `https` URL. We propose s-links, a set of lightweight HTML extensions to express more complex security policies in links such as key pinning. This is the simplest and most efficient way to secure connections to new domains before persistent security policy can be negotiated directly, requiring no changes to the user experience and aligning trust decisions with the user's mental model. We show how s-links can benefit a variety of proposed protocols and discuss implications for the browser's same-origin policy.**

I. Introduction

HTTPS [13], which layers HTTP traffic over SSL/TLS [4] for confidentiality and integrity, is the dominant protocol for securing web traffic. Though there have been many subtle cryptographic flaws in TLS (see [2] for an extensive survey), the greatest concerns are incomplete deployment of HTTPS (enabling *stripping attacks* [10]) and weaknesses in the certificate authority (CA) system by which attackers have obtained certificates for domains they did not own.

The emerging solution to HTTPS stripping is *strict transport security* (HSTS) [7] through which browsers learn that specific domains must only be accessed via HTTPS. HSTS provides a model for scalable, end-to-end distribution of security policy: browser preload HSTS for popular domains, follow HTTPS links to reach new domains securely, and then cache HSTS policies for frequently-visited domains when HSTS headers are observed.

Following several high-profile CA failures and evidence of fraudulent certificates being used in real-world network eavesdropping attacks [15], many protocols have been proposed to maintain security against an attacker with a *rogue certificate* for a target domain signed by a trusted CA. This is a challenging problem, and it remains unclear which of several competing approaches will prove practical.

Any solution though must support varying security policies at different domains. Browsers can preload policies for the largest domains and policy can be established by continuity once a domain is securely reached, but this leaves initial connections to many domains vulnerable. We advocate that in-band secure introduction using the web's existing hyperlink model is the only viable approach to distributing security policy without adding significant communication overhead or changing the user experience.

II. The Strict Transport Security model

We extract several important design principles from the deployment of HSTS to combat stripping attacks, which occur when an active network attacker opportunistically prevents HTTPS traffic by downgrading HTTPS links, blocking HTTPS redirects and proxying traffic to the genuine server. There are now turnkey software packages for performing these attacks, namely *sslstrip* [10]. Browsers historically relied on users to manually detect stripping, but usability studies find that most users don't notice the absence of HTTPS indicators (such as padlock icons) at all [14].

HSTS is a binary (per domain[1]) security policy.[2] Its presence is **invisible to users**, requiring no user decisions for security. It is also **incrementally deployable**, with individual domains gaining protection unilaterally by opting in. HSTS-protected domains can easily perform **secure introduction** of users to other HSTS-protected domains by serving HTTPS links, with no vulnerability to a network attacker stripping the HSTS upgrade for the new domain. Secure introduction enables users to more easily change the parties they rely on for security (**trust agility**) and, critically, to more intuitively understand whom they are trusting when connecting to a new domain (**trust affordance**).

A. Preloaded HSTS

The Chrome browser now ships with a preloaded HSTS policy for approximately 150 domains, with Firefox developing a similar approach. The EFF's HTTPS Everywhere browser extension[3] provides similar protection for a much larger list of over 5,000 domains. Security for preloaded domains reduces to maintaining a genuine, up-to-date browser.[4] Of course, this approach can't scale indefinitely. Based on data from web crawls performed for Google search, over 15,000 domains are now serving HSTS headers. Future growth will likely exceed the number of domains which can be maintained by browser vendors and efficiently checked during every connection.

[1]By default, HSTS is declared for a fully-qualified domain name, though there is an optional `includeSubDomains` directive.

[2]HSTS has other security implications beyond requiring HTTPS, such as preventing users from clicking through certificate error messages.

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

[4]Chrome's preload list has an expiration date if the browser is not regularly updated, to prevent a non-updated browser from losing access to domains if they cease to support HTTPS.

B. *HSTS through continuity*

After a successful HTTPS connection, domains can assert an HSTS policy via an HTTP header. The browser will store the HSTS policy for an arbitrary time period set by the `max-age` directive. The policy can be renewed indefinitely by future connections, maintaining security as long as the domain is visited at least once per `max-age` period. This can be described as a *continuity* protocol,[5] reducing security to a single vulnerable initial connection.[6]

C. *Secure introduction to HSTS domains*

Protecting initial connections requires querying a trusted authority to determine if the new domain supports HTTPS. The authority cannot be the target domain itself, because an adversary can simply block HTTPS traffic and cause the browser to infer that only HTTP is supported. The authority must also be so reliable that browsers can hard-fail if the authority doesn't respond, otherwise the attacker can similarly block traffic to the authority. This reliability requirement makes out-of-band queries prohibitive.[7]

However, if a user navigates to a new HSTS domain from a known HSTS domain, this *introducer* can implicitly serve as a trusted authority by serving an HTTPS link. The link serves as an assertion that the destination domain supports HTTPS and the initial connection must fail otherwise. The browser can then establish the new domain's persistent HSTS status via continuity. Thus, end-to-end security exists under a simplified model of browsing behavior:

> **Linked web navigation model:** *users only reach new domains via hyperlinks, beginning with a set of domains with preloaded security policies.*

This is a crude model which ignores users' ability to type in a new domain's URL directly. However, although a large number of web requests do come from direct navigation (typing) or bookmarks, hyperlinks play a disproportionately large role in discovering new websites, particularly from search engines [12].

HSTS under a linked navigation model requires no user decisions, with all violations resulting in hard failures that are barely distinguishable from non-existent domains. Users trust only their browser vendor and any sites at which links are clicked. These trusted parties can be understood and changed more easily than CAs, DNS servers, or other "invisible" trusted parties, enabling relatively high trust affordance and agility.

III. Proposed security policies

Several security protocols have been proposed to mitigate rogue certificates. We list three approaches here:

A. *Key pinning within certificate chains*

Key pinning specifies a limited set of public keys[8] which a domain can use in establishing a TLS connection. At least one public key in the pin set must appear somewhere in the server's certificate chain, enabling pinning to the server's end-entity public key, the key of the server's preferred root CA, or the key of any intermediate CA. This means rogue certificates must include one of the pinned keys, removing vulnerability to any CA not in the pin set. Similarly to HSTS, Chrome has implemented preloaded pinning for a small number of domains and the HPKP draft specification enables servers to assert pins via HTTP headers [6].

B. *Out-of-chain key pinning*

Another approach 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.[9] TACK [11] proposes distributing out-of-chain public keys using continuity,[10] while Sovereign Keys [5] proposes using a public append-only log.

C. *Public logging*

Certificate Transparency [9] (CT) is a proposal to detect rogue certificates by 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[11] by all public CAs.[12] While not part of the current draft proposal, it's possible that as CT is incrementally deployed supporting domains will wish to assert a "CT-required" security policy through browser preloads or a continuity protocol.

IV. Developing extensible secure links

All of the above proposals suffer from untrusted initial connections because there is no mechanism to declare security policy in links, unlike HSTS which is effectively expressed by HTTPS links.

[5]HSTS can also be described as a "trust-on-first-use" (TOFU) scheme. We use the more general term continuity which can describe protocols like TACK which do not afford full trust upon initial connection.

[6]In addition to true initial connections, any connections after extended inactivity or flushing of the browser's HSTS state will be vulnerable.

[7]The OCSP protocol for checking a certificate's revocation status is instructive. All browsers skip OCSP checks if the server can't be reached, though this allows a network attacker to easily undermine the protocol.

[8]Specifically, a key pinning policy will specify the hash of the complete Subject Public Key Info field of an X.509 certificate.

[9]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.

[10]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.

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

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

A. Security pseudo-protocols and modifying URLs

One approach is to establish new pseudo-protocols for use in URLs, just as `https` supplants `http`. This is straightforward for binary security policies.[13] CT, for instance, could benefit from an `httpsct` pseudo-protocol which requires a CT log signature for any presented certificate. Specifying more complex protocols like key pinning is considerably more difficult. YURLs [3] are a proposal to specify keys in URLs with the `httpsy` protocol,[14] but these are unwieldy to type and will be rejected by legacy clients which can't parse the new pseudo-protocol. URLs have also (perhaps unwisely) taken on a critical role in the security user interface as users are expected to recognize them to determine which domain they are interacting with. Composing security policies (for example, combining key pinning and CT) requires further pseudo-protocols to be developed. Finally, embedded security policy goes against the original philosophy of URLs as the permanent location of network resources. Security policy may evolve or expire while a URL refers to the same content.

B. S-links: security attributes in HTML

We therefore propose s-links,[15] specifying security policy not in URLs but as HTML attributes in tags which indicate connections between documents, such as the `a` (anchor) tag which produces a hyperlink. Unlike URLs, new HTML attributes are ignored by legacy user agents, invisible to users, and easily extensible. They are explicitly not part of a resource's permanent location, but a means to perform secure introduction so that persistent policy can be established by continuity. A minimal example would be the following s-link requiring key pins $\{k_1, k_2\}$ and expiring at time t:

```
<a link-security="expiry=t;
pin-sha256=H(k1); pin-sha256=H(k2);"
href="https://www.example.com">secure</a>
```

We leave details to our draft specification. Remaining open questions include handling redirects and compressing repeated s-link policies for pages with many links.

C. User experience of s-links

To ensure invisibility to end users, violations of s-link directives should produce hard failures with no "click through" option. Savvy users may extract the raw URL, but most will experience a traditional "broken link" and either choose a different introducer or assume the new page is unreachable. We assume, as today, that users trust the source web site when clicking on a link, so a failure of that site's security policy is a hard error.

[13]One such proposal is `httpsev`, requiring HTTPS with an Extended Validation certificate [8].

[14]YURLs are less flexible than the key pinning described by HPKP or TACK, only specifying a single end-entity key hash.

[15]A working specification is provided at www.secure-links.org.

D. Malicious s-links

S-links should provide no new capability to malicious web sites, including legitimate sites compromised by cross-site scripting. To ensure this, s-links may only specify *stricter* security policy than what would otherwise be enforced. This limits malicious s-links to specifying a broken hyperlink, which malicious links may already do.

E. Expiration

Valid s-links must include an `expiry` directive. This prevents cached links from breaking if security policies change. S-links include a specific expiration date (rather than a time to live) because cached HTML can lose its origin date. In practice, sites will discover policies to include in s-links through continuity-based scanning[16] of destination domains, which should include an explicit expiration. For example, the `max-age` directive in HPKP headers can be translated into the `expiry` field of an s-link specifying key pins.

V. Secure links and the same-origin policy

Browsers isolate content using the *same-origin policy*, where the origin is defined as the scheme, host, and port of the content's URL. This means content delivered over HTTPS is isolated from any insecure HTTP content an attacker injects with the same host and port. However, content delivered over HTTPS but with different s-link directives will represent separate *fine-grained origins* [8] which are not isolated by the same-origin policy.

Without further consideration, this could enable an attacker to undermine content reached via an s-link. For example, if a user visiting `https://secure.com` follows an s-link to `https://target.com`, an attacker with a rogue certificate could inject an unprotected malicious frame for `https://target.com` and potentially use it to inject scripts [8] or initiate cross-frame navigation [1] of the securely-loaded frame.[17] A network attacker could insert the malicious frame in real-time after observing the s-link-protected request and ensure it loads before the protected frame can negotiate a security upgrade.

One approach is to extend the same-origin policy [8] to include security policies used to load each frame. However, this approach can prevent valid intra-origin communication as any variation in s-link policies would fork content into isolated domains.

Crucially, s-links have a limited goal of reaching new domains that will immediately negotiate a persistent security upgrade (such as setting key pins via HPKP headers). For this use case, there is no need to support content loaded from the same origin under different security policies, only to

[16]We can't expect all introducer domains to perform their own scanning. Security policies may also be cached by public network notaries.

[17]These attacks require the attacker-controlled frame to have a JavaScript reference to the secure frame. This can occur if the secure frame is given a global name via `window.open`.

track origin-wide security policy as it evolves. For example, a crude but effective policy would be to flush all cached resources whenever a domain's security policy is made more restrictive. This would include (at minimum) reloading any open frames, evicting any HTTP-cached content, removing cookies, deleting HTML5 localStorage and AppCache contents, and dropping any TLS session IDs. This would extirpate any latent attacker-controlled resources.

A refinement of this approach is *retroactive policy checking*. After a domain's persistent security policy changes, previously-loaded resources must be evicted only if they were loaded over a TLS connection which would violate the new policy. For example, when key pins are set, content previously loaded under keys not in the new pin set must be evicted. Because most domains will only mandate a policy that they have already been complying with, retroactive checking should rarely cause eviction in normal use.

In fact, retroactive policy checking is useful not only for s-links, but to maintain security any time policy is upgraded, such as updates to the browser's preloaded policies. It is less critical for non s-link protected continuity-based policy upgrades because an attacker who can impersonate the intended domain can prevent the upgrade altogether during the unprotected initial connection.

A final subtlety is that eviction of active content (scripts) must be atomic. That is, all active resources must be evicted before any are reloaded. In the absence of this requirement, the last potentially-malicious script to be removed could maintain its presence by attacking newly delivered content.

A. *Secure resource loading*

S-link attributes can be included in tags like `script`, enabling secure frames to load non-framed resources such as JavaScript libraries, images, or CSS from third-party domains which don't set persistent security policy. Unlike framed content, these resources inherit the origin of the parent frame and hence will be inaccessible to attackers who can impersonate the unprotected third-party domain.

VI. Discussion and concluding remarks

Hardening the web against rogue certificates will require the gradual deployment of policies stricter than today's permissive global HTTPS policy. Based on the experience of HSTS we argue that a combination of browser-preloaded policies, continuity-based policy negotiation, and secure introduction can provide a solution which is incrementally deployable, reflects users' current trust model, and provides end-to-end security within a linked navigation model.

This blueprint remains years away from reality. It will require further work by browser vendors to develop a sustainable approach to preloading security policy for critical domains and protecting the browser update process. It will require effort by webmasters to declare support for new security policies; HSTS suggests diffusion of new protocols will take many years. Finally it depends on major hubs on the web growing into a new role as notaries of security policy around the web so that they can perform secure introduction. Today's search engines don't yet consistently provide HTTPS links to HSTS domains.

Still, we consider this the most likely path forward. Changing the default HTTPS policy for the web appears impossible, meaning domain-specific policies must be distributed. Adding network connections to every web request to fetch security policy is a daunting hurdle. Therefore we believe secure links are the most practical approach to distributing new security policies as they become available. They are also the most natural extensions of the web's traditionally decentralized structure and provide the best hope of user-visible trust agility.

Acknowledgements

I would like to thank Trevor Perrin, Jeremy Clark, Ryan Sleevi, Adam Langley, Umesh Shankar, Chris Palmer, Adam Barth and Phil Ames for discussions which led to this paper.

References

[1] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. *Communications of the ACM*, 52(6):83–91, 2009.

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

[3] T. Close. YURLs. http://www.waterken.com/dev/YURL/, 2004.

[4] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, May 2000.

[5] P. Eckersley. Internet-Draft: Sovereign Key Cryptography for Internet Domains. 2012.

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

[7] J. Hodges, C. Jackson, and A. Barth. RFC 6797: HTTP Strict Transport Security (HSTS). 2012.

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

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

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

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

[12] M. R. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking web sites with real user traffic. WSDM '08. ACM, 2008.

[13] E. Rescorla. HTTP over TLS. RFC 2818, 2000.

[14] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

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