Beyond the Basics of HTTPS Serving

ADAM LANGLEY

Adam Langley is a software engineer working on Google Chrome.
agl@chromium.org

A number of factors are contributing to an increase in the number of Web sites that choose to deploy HTTPS. Tools such as Firesheep [1] have highlighted the vulnerability of the transport layer now that hosts are often connected via public, wireless networks; at the much larger scale, there have been several instances of national-level actors performing countrywide attacks [2, 3].

As the incentives to use HTTPS have increased, the costs have decreased. CPU time spent on SSL is a diminishing fraction for increasingly complex Web sites, even as processing power becomes cheaper.

None of these forces seem likely to change, so running an HTTPS site is increasingly likely to be part of your job in the future, if it isn't already.

The Stripping Problem

When entering a URL into a browser's address bar, few users bother to enter any scheme part at all. Since the default scheme is HTTP, they are implicitly requesting an insecure connection. Although sites which use HTTPS pervasively will immediately redirect them to an HTTPS URL, this gap is all an attacker needs. By stopping the redirect, an attacker can proxy the real site over HTTP and effectively bypass HTTPS entirely. Very observant users may notice the lack of security indications in their browser, but this is generally a very effective attack that is commonly known as SSL stripping, after the demonstration tool by Moxie Marlinspike of the same name [4].

HTTP Strict Transport Security (HSTS) [5] allows a Web site to opt in to being HTTPS only. For an HSTS site, a browser will only send HTTPS requests, eliminating the window of insecurity. HSTS is currently an IETF draft, but has already been implemented by both Chrome and Firefox.

Web sites opt into HSTS by means of an HTTP header, such as:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

HSTS headers are only accepted over HTTPS to stop denial-of-service attacks. The HSTS property is cached for the given number of seconds (about a year in this example) and, optionally, also for all subdomains of the current domain. (Including subdomains is highly recommended, as it stops an attacker from directing a user to a subdomain over HTTP and capturing any domain-wide, insecure cookies.)

HSTS also makes certificate errors fatal for the site in question. As the Web has penetrated into everyday life, users are now ordinary people, and asking them to

evaluate complex certificate validity questions is ridiculous. Fatal errors do, of course, make it critical that you renew certificates in a timely fashion, but remaining certificate validity should be part of any monitoring setup. It's also important that the names in your certificate match what you actually use. Remember that a wildcard matches exactly one label. So *.example.com doesn't match example.com or foo.bar.example.com. Usually you want to request a certificate from your CA that includes example.com and *.example.com.

As HSTS properties are learned on the first visit, there's still a gap before a user has visited a site for the first time where HSTS isn't in effect. This gap is also a problem after the user has cleared browsing history. In order to counter this, Chrome has a built-in list of HSTS sites which is always in effect. High-security sites are invited to contact the author in order to be included. Over time this list may grow unwieldy, but that's a problem that we would love to have. At the moment the list is just over 60 entries, although that does include sites such as Gmail, Paypal, and Twitter.

Mixed Scripting

Mixed scripting is a subset of the more general problem of mixed content. Mixed content arises whenever an HTTPS origin loads resources over an insecure transport. Since the insecure resources cannot be trusted, the security of the whole page is called into question.

The impact of a mixed content error depends on the importance of the resource in question. In the case of an insecure image, the attacker can only control that image. However, in the case of JavaScript, CSS, and embedded objects, the attacker can use them to take control of the whole page and, due to the same-origin policy, any page in that origin. When these types of resources are insecure, we call it mixed scripting because of the greatly increased severity of the problem.

Browsers typically inform the user of mixed scripting in some fashion, either by removing the security indicators that usually come with HTTPS, or by highlighting the insecurity itself. Chrome will track the problem across pages within the same origin, which yields a more accurate indication of which pages are untrustworthy, even though this has confused some developers.

But years of nasty warnings have failed to solve the problem, and mixed scripting is an insidious threat for sites that mix HTTP and HTTPS. Such sites will often have HTTP pages that are not expected to be served over HTTPS, but which are accessible as such. Since no normal interaction will lead to the HTTPS version, no warnings will ever appear. But with mixed scripting the attacker gets to choose the location. By injecting an iFrame or a redirect into any HTTP request (to any site), an attacker can cause a browser to load a given HTTPS page. By picking a page with mixed scripting, they can then hijack the insecure requests for resources and compromise the origin.

The real solution is for browsers to block mixed script loads. Internet Explorer version 9 already does this and Chrome will soon, so it's past time to pay attention to any mixed script warnings before your site breaks. But for other browsers, which will still be dominant for many years to come, you have to make sure that there's no mixed scripting on any of your pages, by actively searching for them in the same way that an attacker would (although some respite may come in the form of CSP, which we'll consider next).

Content Security Policy

I will only briefly reference CSP [6] here, as it's a large topic and one that strays outside of transport security. However, it is being implemented in both WebKit and Firefox, and it allows a site, by means of another HTTP header, to limit the origins from which active resources can be sourced. By using it to eliminate any non-HTTPS origins, mixed scripting can be solved for the set of browsers that implement CSP but don't block by default.

CSP also has the ability to send reports back when it encounters a violation of the policy. By monitoring these reports, a site can discover mixed-scripting errors that real-world users are experiencing.

Secure Cookies

For sites that aren't HSTS, it's important that they mark any sensitive cookies as "secure." By default, the same origin policy for cookies does not consider port or protocol [11]. So any insecure requests to the same domain will contain the full set of cookies in the clear, where an attacker can capture them. Without HSTS, users are likely to make HTTP requests even to HTTPS-only sites when they enter a domain name to navigate.

Cookies marked as secure will only be sent over HTTPS. Even if your site is using HSTS, with all subdomains included, you should still mark your sensitive cookies as secure, simply as a matter of good practice.

DNSSEC and Certificate Verification

Once you have all of the above sorted out, you may want to worry about how certificate verification works. The number of root certificate authorities trusted by Windows, OS X, or Firefox may provoke worry in some quarters, but, due to an unfortunate legacy, the situation is rather worse.

Normal certificates, used by sites for HTTPS, are end-entity certificates. They contain a public key which is used to provide transport security for a host, and that public key can't be used to sign other certificates. But there are a number of cases where organizations want an intermediate CA certificate, one that can be used to sign other certificates. There do exist mechanisms for limiting the power of intermediate CA certificates, but they are far from universally supported by clients, which will often reject such certificates. This leads to intermediate CA certificates typically being issued with the full signing power of the root CA that issues them.

Although the root CA is technically responsible for them, the identities of intermediate CA holders are not public, nor are the handling requirements that the root CA imposes on them. Thanks to stellar work by the EFF [7], which crawled the Web for the subset of intermediate CAs that have issued public Web site certificates, we know that there are at least 1,482 such certificates held by an estimated 651 different organizations.

This has prompted questions about the foundations of HTTPS and several efforts to address the problem. Probably the most prominent area of focus for solutions involves using DNSSEC, either as an alternative or as a constraint.

As a PKI, DNSSEC has much to commend it. Although it's certainly not simple, it is much less complex than PKIX (the standard for existing certificates). It inherently

solves the intermediate CA problem, because of DNS's hierarchical nature. It may also encourage the use of HTTPS, because Web sites must already have a relationship with a DNS registrar, who can also provide DNSSEC.

Most existing certificates are based on proving ownership of a DNS name, often via email. These are called Domain Validation (DV) certificates and they fundamentally rest on DNS. A compromise of your DNS can easily be turned into a DV certificate via an existing certificate authority, so DNSSEC has a good claim to be at least as strong as DV certificates. (Extended Validation (EV) certificates are significantly more rigorous and are not candidates for any of the measures described here.)

With DNSSEC in hand, there are two broad categories of statements that we might want to make about certificates. We might wish to exclude existing certificates ("This site's CA is X, accept nothing else"), or we might wish to authorize certificates that wouldn't otherwise be valid ("X is a valid public key for this site"). On another axis, we might want these statements to be processed by clients or by CAs.

In the "exclusion" and "processed by CAs" corner, we have CA Authorization (CAA) records [8]. These DNS records are designed to inform a CA whether they are authorized to issue a certificate in a given domain. Although CAs will check DNS ownership in any case, this provides a useful second line of defense, and, since there aren't many root CAs that issue to the general public, deployment is a tangible goal in the medium term. Additionally, since DNSSEC is signature based, CAs can retain the DNSSEC chain that they resolved as a proof of correctness in the event of a dispute.

Covering both types of statements designed to be processed by clients is DANE [9], and this is where much of the work is occurring. Although DANE is by no means designed exclusively for browsers, browsers are the dominant HTTPS client on the Internet at the moment and much of the discussion tends to start with them.

On that basis, it's worth considering some of the headwinds that any DNSSEC solution designed for browsers to implement will face.

First, DNSSEC resolution ability on the client is almost non-existent at the moment. Without this, browsers would be forced to ship their own DNSSEC resolver libraries, which is a degree of complexity and bloat that we would really rather avoid. Even assuming that the client is capable of resolution, DNS is probably the most adulterated protocol on the Internet; it seems that every cheap firewall and hotel network abuses and filters it. In an experiment conducted by a large population of consenting Chrome users, around 0.5–1% of DNS requests for a random DNS resource record type (13172) failed to get any reply, even after retries, for domain names that were known to exist. Based on this, any DNSSEC resolution will have to assume a significant amount of filtering and misbehavior of the network when faced with DNSSEC record types.

These troubles suggest that any certificate exclusion will have a very troubled deployment as, in order to be effective, exclusion has to block on getting a secure answer. An exclusion scheme that can be defeated by filtering DNS requests is ineffective. Even DNSSEC-based certificate authorization would be unreliable and frustrating.

Setting aside the functionality problems for the moment, performance is also a concern. In the same experiment described above, 2.5% of the replies that were

received arrived over 100 ms after the browser had set up a TCP connection to, performed a TLS handshake with, and verified the certificate of the same domain name. For authorization, the site bears the performance impact and so that, at least, is tenable. For exclusion, with its blocking lookup, these penalties would be imposed on every site.

Although DNSSEC resolution on the client would appear to face significant hurdles, there are still options. First, for exclusion we could sacrifice the absolute guarantees and use a learning scheme,as HSTS does. In this design, the lookups would proceed asynchronously, but the results would be persisted by the browser in order to protect future connections. This is workable, but a similar scheme that used HSTS-like HTTP headers would be able to achieve the same results with dramatically reduced complexity.

Finally, as DNSSEC is signature based, there's no reason why DNSSEC records have to be transported using the DNS protocol. If the correct data and signatures can be delivered in another fashion, they can be verified just as well. So, as an experiment, Chrome accepts a form of self-signed certificate that carries a DNSSEC chain proving its validity [10], effectively adding DNSSEC as another root CA. Since it'll be several decades before any new form of certificate can achieve 99 or even 95 percent acceptance in browsers, there's no chance of major sites using them. But there are many HTTPS sites on the Internet that don't currently have a valid certificate and this may be attractive for them. We'll be evaluating the level of support in twelve months and considering whether to continue the experiment.

In conclusion, there are several modern developments that HTTPS sites should be planning and implementing right now. If nothing else, I highly recommend using the HTTPS scanner at https://ssllabs.com to check your sites for any configuration errors. Securing cookies and fixing mixed scripting is essential, and for sites that are exclusively HTTPS, HSTS should be implemented. Certificate validation is likely to see some changes in the coming years and it's worth keeping an eye open, even if there's no immediate action needed for most sites.

References

[1] http://codebutler.com/firesheep.

[2] http://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook.

[3] https://www.eff.org/deeplinks/2011/08/iranian-man-middle-attack-against-google.

[4] http://www.thoughtcrime.org/software/sslstrip/.

[5] http://tools.ietf.org/html/draft-hodges-strict-transport-sec.

[6] https://wiki.mozilla.org/Security/CSP/Specification.

[7] https://www.eff.org/observatory.

[8] http://tools.ietf.org/html/draft-hallambaker-donotissue-04.

[9] http://tools.ietf.org/html/draft-ietf-dane-protocol-11.

[10] http://www.imperialviolet.org/2011/06/16/dnssecchrome.html.

[11] http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_cookies.