How To Migrate To HTTPS

By Chris Palmer, last updated 23 January 2015

This document describes a series of steps you can follow to gradually migrate a small or large web site from HTTP to HTTPS. You can take the steps at any speed — all in 1 day, or 1 step per month — but you should take them in order.

Each step is an incremental improvement and has value on its own. However, completing all the migration steps is a necessary (though insufficient) precondition for your site to provide a security guarantee to your users.

Thanks to Cesar Andreu, Andrew Ayer, Yaw Boakye, Bram van Damme, Pierre Far, Matt Gaunt, Romain Guerin, Rafael Lins, Mark Nottingham, Addy Osmani, Gabriel de Perthuis, Jesse Ruderman, Dion Williams, and Jeffrey Yasskin for helpful contributions. Any errors or omissions are due to me. If you see any errors or omissions in this document, please let me know by email (chris@noncombatant.org).

Who Should Read This Document?

This document is for site operators — operations staff, software developers, devops, and their managers — who operate sites that are currently HTTP-only and who want to migrate to (or at least support) HTTPS. There are many reasons to migrate to HTTPS, but for the sake of brevity I'll assume you've already decided you want to.

1: Get And Install Certificates

If you don’t already have certificates for your site’s hostname(s), go to sslmate.com and buy some. You have 2 options:

- Standard: $16/year, valid for example.com and www.example.com.
- Wildcard: $150/year. valid for example.com and *.example.com.

Obviously, Wildcard is economical when you have more than 9 subdomains; otherwise, you can just buy 1 or more Standard certificates. (If you have more than, say, 5 subdomains, you might find a wildcard certificate more convenient when you get to Step 2.)

> **NOTE:** I mention SSLMate because it automates aspects of certificate purchase and provisioning that people have historically found confusing, and because the price is low. Another low-cost certificate vendor is Namecheap ($9 per year, or $94 per year for a wildcard certificate). In the near future, you won’t have to pay any money at all, thanks to the Let’s Encrypt project. I don’t intend to endorse particular certificate vendors, only to show that getting certificates can be easy and low in cost.
>
> **NOTE:** Keep in mind that in wildcard certificates the wildcard applies to only 1 DNS label. A certificate good for *.example.com will work for foo.example.com and bar.example.com, but *not* for foo.bar.example.com.

Copy the certificates to all your front-end servers in a reasonable place such as `/etc/ssl` (Linux and Unix) or wherever IIS wants them (Windows).

2: Enable HTTPS On Your Servers

At this step, you need to make a crucial decision:

- either use IP-based hosting, with each hostname given its own IP address(es); or
- defer this step until your users stop using IE on Windows XP and Android earlier than 2.3; or
- decide that you do not need to support clients running IE on Windows XP or Android earlier than 2.3.

If you have been using IP-based hosting all along, great! Go ahead. However, most site operators use name-based virtual hosting to conserve IP addresses and because it's more convenient in general. The problem with IE on Windows XP and ancient Android is that they do not understand Server Name Indication (SNI), which is crucial for HTTPS name-based virtual hosting.

Someday — hopefully soon — clients that don't support SNI will all be replaced with modern software. Then we can use name-based virtual hosting with HTTPS, and not lose any users. Monitor the user agent string in your request logs to know when enough of your user population has migrated to modern software. (You can decide what your threshold is; perhaps < 5%, or < 1%, or something.)

If you don't already have HTTPS service available on your servers, turn it on now. Configure your web server to use the certificates you bought and installed. You might find Mozilla's handy configuration generator or SSLMate's dandy configuration generator useful.

If you have many hostnames/subdomains, they'll each need to use the right certificate. (If you have many subdomains, a wildcard certificate might be more convenient.)

> **NOTE:** Many site operators have already completed Steps 1 and 2, but are using HTTPS for the sole purpose of redirecting clients back to HTTP. If you are doing that, stop doing that now. See Step 3 to make sure HTTPS and HTTP work smoothly.
>
> **NOTE:** Ultimately you should redirect HTTP requests to HTTPS and use Strict Transport Security. This is not the right stage in the migration process to do that; see Steps 4 and 5.

Now, and throughout your site's lifetime, check your HTTPS configuration with Qualys' handy SSL Server Test. Your site should score an A or A+; treat anything that causes a lower grade as a bug. (Today's A is tomorrow's B, because attacks against algorithms and protocols always get better!)

3: Make Intra-Site URLs Relative

Now that you are serving your site on both HTTP and HTTPS, it should work as smoothly as possible regardless of protocol. But, a problem arises: mixed content. When you serve a page via HTTPS that includes HTTP resources, browsers will warn the user that the full strength of HTTPS has been lost.

In fact, in the case of active mixed content (script, plug-ins, CSS, iframes), browsers often simply won't load or execute the content at all — resulting in a broken page.

> **NOTE:** It is perfectly OK to include HTTPS resources in an HTTP page.

Additionally, when you link to other pages in your site, users could get downgraded from HTTPS to HTTP.

These problems happen when your pages include fully-qualified, intra-site URLs that use the `http://` scheme. You should change content like this:

```
<h1>Welcome To Example.com</h1>
<script src="http://example.com/jquery.js"></script>
<link rel="stylesheet" href="http://assets.example.com/style.css"/>
<img src="http://img.example.com/logo.png"/>
<p>Read this nice <a href="http://example.com/2014/12/24/">new post
on cats!</a></p>
<p>Check out this <a href="http://foo.com/">other cool site.</a></p>
```

to something like this:

```
<h1>Welcome To Example.com</h1>
<script src="//example.com/jquery.js"></script>
<link rel="stylesheet" href="//assets.example.com/style.css"/>
<img src="//img.example.com/logo.png"/>
<p>Read this nice <a href="//example.com/2014/12/24/">new post on
cats!</a></p>
<p>Check out this <a href="http://foo.com/">other cool site.</a></p>
```

or this:

```
<h1>Welcome To Example.com</h1>
<script src="/jquery.js"></script>
<link rel="stylesheet" href="//assets.example.com/style.css"/>
<img src="//img.example.com/logo.png"/>
<p>Read this nice <a href="/2014/12/24/">new post on cats!</a></p>
<p>Check out this <a href="http://foo.com/">other cool site.</a></p>
```

That is, make intra-site URLs as relative as possible: either protocol-relative (lacking a protocol, starting with `//example.com`) or host-relative (starting with just the path, like `/jquery.js`).

> **NOTE:** Do this with a script, not by hand. If your site's content is in a database, you'll want to test your script on a development copy of your database. If your site's content is simple files, test your script on a development copy of the files. Only push the changes to production after the changes pass QA, as normal. You can use Bram van Damme's script or something like it to detect mixed content in your site.
>
> **NOTE:** When linking to other sites (as opposed to including resources from them), don't change the protocol since you don't have control over how those sites operate.
>
> **NOTE:** I recommend protocol-relative URLs to make migration smoother for large sites. If you are not sure you can fully deploy HTTPS yet, forcing your site to use HTTPS for all sub-resources may backfire. There is likely to be a period of time in which HTTPS is new and weird for you, and the HTTP site must still work as well as ever. Over time, you'll complete the migration and can lock in HTTPS (see Steps 4 and 5).

If your site depends on script, image, or other resources served from a third party, such as a CDN, jquery.com, or the like, you have 2 options:

- Use protocol-relative URLs for these resources, too. If the third party does not serve HTTPS, ask them to. Most already do, including jquery.com.
- Serve the resources from a server you control, and which offers both HTTP and HTTPS. This is often a good idea anyway, because then you have better control over your site's appearance, performance, and security — you don't have to trust a third party, which is always nice.

Keep in mind also that you will need to change intra-site URLs in your stylesheets, JavaScript, redirect rules, `<link ...>` tags, and CSP declarations as well — not just the HTML pages!

4: Redirect HTTP To HTTPS

Set `<link rel="canonical" href="https://..."/>` tags in your pages. This helps search engines know the best way to get to your site.

Most web servers offer a simple redirect feature; here's how to do it with Apache and here's how to do it with nginx. Use 301 (Moved Permanently) to indicate to search engines and browsers that the HTTPS version is canonical.

5: Turn On Strict Transport Security And Secure Cookies

At this point, you are ready to “lock in” the use of HTTPS. First, use Strict Transport Security to tell clients that they should always connect to your server via HTTPS, even when following an `http://` reference. This defeats attacks such as SSL Stripping, and also avoids the round-trip cost of the 301 redirect we enabled in Step 4.

> **NOTE:** Clients that have noted your site as a Known HSTS Host are likely to *hard-fail* if your site ever has an error in its TLS configuration (such as an expired certificate). Be sure that you are truly ready to run an HTTPS-only site before turning on HSTS.

Turn on HTTP Strict Transport Security by setting the Strict-Transport-Security header. OWASP’s HSTS page has links to instructions for various server software.

Most web servers offer a similar ability to add custom headers.

> **NOTE:** max-age is measured in seconds. You can start with low values and gradually increase the max-age as you become more comfortable operating an HTTPS-only site.

It is also important to make sure that clients never send cookies (such as for authentication or site preferences) over HTTP. For example, if a user’s authentication cookie were to be exposed in plaintext, the security guarantee of their entire session would be destroyed — even if you have done everything else right!

Therefore, change your web application to always set the Secure flag on cookies that it sets. This OWASP page explains how to set the Secure flag in several application frameworks. Every application framework has some way to set the flag.

Migration Concerns

This section discusses concerns operators may have about migrating to HTTPS.

Search Ranking

Google is using HTTPS as a positive search quality indicator. Google also publishes a guide for how to transfer, move, or migrate your site while maintaining its search rank. Bing also publishes guidelines for webmasters.

Performance

When the content and application layers are well-tuned (see Steve Souders' books for great advice), the remaining TLS performance concerns are generally small, relative to the overall cost of the application. Additionally, you can reduce and amortize those costs. (For great advice on TLS optimization and generally, see *High Performance Browser Networking* by Ilya Grigorik.) See also Ivan Ristic's *OpenSSL Cookbook* and *Bulletproof SSL And TLS*.

In some cases, TLS can *improve* performance, mostly as a result of making HTTP/2 possible. I gave a talk on HTTPS and HTTP/2 performance at Chrome Dev Summit 2014.

Referer Headers

User agents will not send the Referer header when users follow links from your HTTPS site to other HTTP sites. If that is a problem, there are several ways to solve it:

- The other sites should migrate to HTTPS. Perhaps they might find this guide useful! `:)` If referee sites can complete Step 2 of this guide, you can change links in your site to theirs from `http://` to `https://`, or you can use protocol-relative links.
- You can use the new Referrer Policy standard to work around a variety of problems with Referer headers.

Because search engines are migrating to HTTPS, you are likely see *more* Referer headers when you migrate to HTTPS than you are now.

According to the HTTP RFC:

> Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol.

Ad Revenue

Site operators that monetize their site by showing ads want to make sure that migrating to HTTPS does not reduce ad impressions. But, due to mixed content security concerns, an

HTTP iframe will not work in an HTTPS page. There is a tricky collective action problem here: until advertisers publish over HTTPS, site operators cannot migrate to HTTPS without losing ad revenue; but until site operators migrate to HTTPS, advertisers have little motivation to publish HTTPS.

Advertisers should at least offer ad service via HTTPS (such as by completing Step 2 in this guide). Many already do. You should ask advertisers that do not serve HTTPS at all to at least start. You may wish to defer completing Step 4 in this guide until enough advertisers interoperate properly.