What does a PHP developer need to know about https / secure socket layer connections?

Asked 16 years, 3 months ago Modified 7 years, 9 months ago

Viewed 6k times Part of PHP Collective

10

I know next to nothing when it comes to the how and why of https connections. Obviously, when I'm transmitting secure data like passwords or especially credit card information, https is a critical tool. What do I need to know about it, though? What are the most common mistakes you see developers making when they implement it in their projects? Are there times when https is just a bad idea? Thanks!

PHP php security ssl https

Share

Improve this question

Follow

asked Sep 15, 2008 at 16:51

user5564

In addition to the below, you should use Wireshark to look at a simple HTTPS request/response just to see it "in action." And it's implicit in the answers below that you understand public key cryptography (at a high level). – Andrew Coleson Jan 31, 2009 at 19:20

7 Answers

Sorted by: Highest score (default)

22

An HTTPS, or Secure Sockets Layer (SSL) certificate is served for a site, and is typically signed by a Certificate Authority (CA), which is effectively a trusted 3rd party that verifies some basic details about your site, and certifies it for use in browsers. If your browser trusts the CA, then it trusts any certificates signed by that CA (this is known as the trust chain).

Each HTTP (or HTTPS) request consists of two parts: a request, and a response. When you request something through HTTPS, there are actually a few things happening in the background:

- The client (browser) does a "handshake", where it requests the server's public key and identification.
    - At this point, the browser can check for validity (does the site name match? is the date range current? is it signed by a CA it trusts?). It can even contact the CA and make sure the certificate is valid.

- The client creates a new pre-master secret, which is encrypted using the servers's public key (so only the server can decrypt it) and sent to the server
- The server and client both use this pre-master secret to generate the master secret, which is then used to create a symmetric session key for the actual data exchange
- Both sides send a message saying they're done the handshake
- The server then processes the request normally, and then encrypts the response using the session key

If the connection is kept open, the same symmetric key will be used for each.

If a new connection is established, and both sides still have the master secret, new session keys can be generated in an 'abbreviated handshake'. Typically a browser will store a master secret until it's closed, while a server will store it for a few minutes or several hours (depending on configuration).

For more on the length of sessions see How long does an HTTPS symmetric key last?

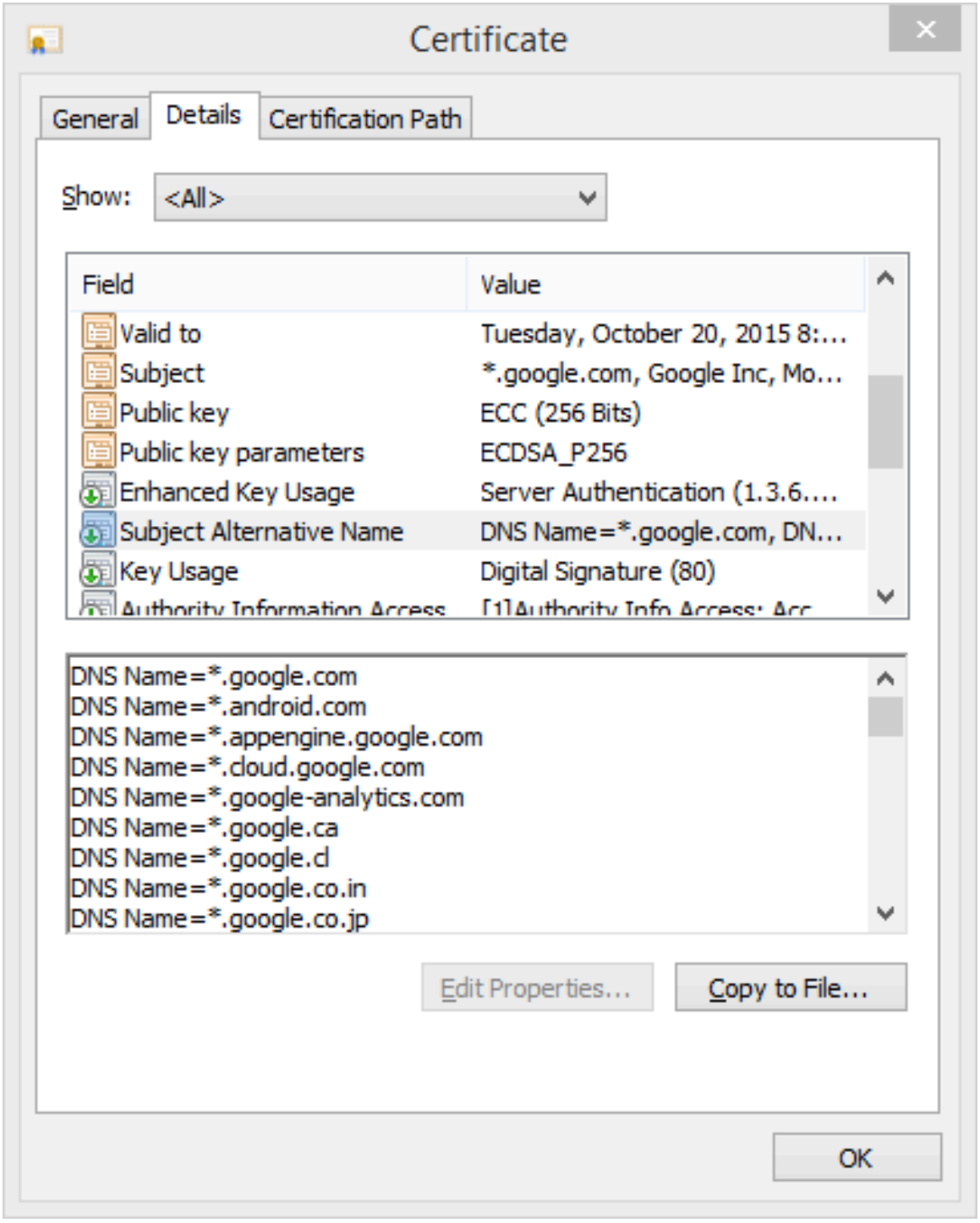
Certificates and Hostnames

Certificates are assigned a Common Name (CN), which for HTTPS is the domain name. The CN has to match exactly, eg, a certificate with a CN of "example.com" will

NOT match the domain "www.example.com", and users will get a warning in their browser.

Before SNI, it was not possible to host multiple domain names on one IP. Because the certificate is fetched before the client even sends the actual HTTP request, and the HTTP request contains the Host: header line that tells the server what URL to use, there is no way for the server to know what certificate to serve for a given request. SNI adds the hostname to part of the TLS handshake, and so as long as it's supported on both client and server (and in 2015, it is widely supported) then the server can choose the correct certificate.

Even without SNI, one way to serve multiple hostnames is with certificates that include Subject Alternative Names (SANs), which are essentially additional domains the certificate is valid for. Google uses a single certificate to secure many of it's sites, for example.



Another way is to use wildcard certificates. It is possible to get a certificate like "*.example.com" in which case "www.example.com" and "foo.example.com" will both be valid for that certificate. However, note that "example.com" does not match "*.example.com", and neither does "foo.bar.example.com". If you use "www.example.com" for your certificate, you should redirect anyone at "example.com" to the "www." site. If they request https://example.com, unless you host it on a

separate IP and have two certificates, the will get a certificate error.

Of course, you can mix both wildcard and SANs (as long as your CA lets you do this) and get a certificate for both "example.com" and with SANs "*.example.com", "example.net", and "*.example.net", for example.

Forms

Strictly speaking, if you are submitting a form, it doesn't matter if the form page itself is not encrypted, as long as the submit URL goes to an https:// URL. In reality, users have been trained (at least in theory) not to submit pages unless they see the little "lock icon", so even the form itself should be served via HTTPS to get this.

Traffic and Server Load

HTTPS traffic is much bigger than its equivalent HTTP traffic (due to encryption and certificate overhead), and it also puts a bigger strain on the server (encrypting and decrypting). If you have a heavily-loaded server, it may be desirable to be very selective about what content is served using HTTPS.

Best Practices

- If you're not just using HTTPS for the entire site, it should automatically redirect to HTTPS as required. Whenever a user is logged in, they should be using HTTPS, and if you're using session cookies, the

cookie should have the secure flag set. This prevents interception of the session cookie, which is especially important given the popularity of open (unencrypted) wifi networks.

- Any resources on the page should come from the same scheme being used for the page. If you try to fetch images from http:// when the page is loaded with HTTPS, the user will get security warnings. You should either use fully-qualified URLs, or another easy way is to use absolute URLs that do not include the hostname (eg, src="/images/foo.png") because they work for both.
    - This includes external resources (eg, Google Analytics)
- Don't do POSTs (form submits) when changing from HTTPS to HTTP. Most browsers will flag this as a security warning.

Share Improve this answer Follow

edited Mar 21, 2017 at 14:55

answered Sep 15, 2008 at 22:46

gregmac
25.2k 11 90 120

The client neither sends its public key nor generates one on the fly. SSL uses symmetric encryption, not asymmetric encryption (PKI). – user207421 Mar 20, 2017 at 1:33

@EJP You are right, I did an update to that part. Knowing what I know now (it's been 8 years since I wrote that) this is

only really just the surface of the topic and there's a ton of nuance with SSL keys and handshakes that's not covered (eg: different key exchange algorithms, layer 4 vs layer 7 load balancers and ssl offloading vs bridging). – gregmac Mar 21, 2017 at 15:04

5

I'm not going to go in depth on SSL in general, gregmac did a great job on that, see below ;-).

However, some of the most common (and critical) mistakes made (not specifically PHP) with regards to use of SSL/TLS:

1. Allowing HTTP when you should be enforcing HTTPS
2. Retrieving some resources over HTTP from an HTTPS page (e.g. images, IFRAMEs, etc)
3. Directing to HTTP page from HTTPS page unintentionally - note that this includes "fake" pages, such as "about:blank" (I've seen this used as IFRAME placeholders), this will needlessly and unpleasantly popup a warning.
4. Web server configured to support old, unsecure versions of SSL (e.g. SSL v2 is common, yet horribly broken) (okay, this isn't exactly the programmer's issue, but sometimes noone else will handle it...)
5. Web server configured to support unsecure cipher suites (I've seen NULL ciphers only in use, which basically provides absolutely NO encryption) (ditto)

6. Self-signed certificates - prevents users from verifying the site's identity.
7. Requesting the user's credentials from an HTTP page, even if submitting to an HTTPS page. Again, this prevents a user from validating the server's identity BEFORE giving it his password... Even if the password is transmitted encrypted, the user has no way of knowing if he's on a bogus site - or even if it WILL be encrypted.
8. Non-secure cookie - security-related cookies (such as sessionId, authentication token, access token, etc.) **MUST** be set with the "secure" attribute set. This is important! If it's not set to secure, the security cookie, e.g. SessionId, can be transmitted over HTTP (!) - and attackers can ensure this will happen - and thus allowing session hijacking etc. While you're at it (tho this is not directly related), set the HttpOnly attribute on your cookies, too (helps mitigate some XSS).
9. Overly permissive certificates - say you have several subdomains, but not all of them are at the same trust level. For instance, you have www.yourdomain.com, dowload.yourdomain.com, and publicaccess.yourdomain.com. So you might think about going with a wildcard certificate.... BUT you also have secure.yourdomain.com, or finance.yourdomain.com - even on a different server. publicaccess.yourdomain.com will then be able to impersonate secure.yourdomain.com.... While there

may be instances where this is okay, usually you'd want some separation of privileges...

That's all I can remember right now, might re-edit it later...

As far as when is it a BAD idea to use SSL/TLS - if you have public information which is NOT intended for a specific audience (either a single user or registered members), AND you're not particular about them retrieving it specifically from the proper source (e.g. stock ticker values MUST come from an authenticated source...) - then there is no real reason to incur the overhead (and not just performance... dev/test/cert/etc).

However, if you have shared resources (e.g. same server) between your site and another MORE SENSITIVE site, then the more sensitive site should be setting the rules here.

Also, passwords (and other credentials), credit card info, etc should ALWAYS be over SSL/TLS.

answered Sep 16, 2008 at 8:46

AviD

13.1k ● 7 ● 64 ● 93

1

Be sure that, when on an HTTPS page, all elements on the page come from an HTTPS address. This means that elements should have relative paths (e.g. "/images/banner.jpg") so that the protocol is inherited, or that you need to do a check on every page to find the protocol, and use that for all elements.

NB: This includes all outside resources (like Google Analytics javascript files)!

The only down-side I can think of is that it adds (nearly negligible) processing time for the browser and your server. I would suggest encrypting only the transfers that need to be.

Share Improve this answer

Follow

answered Sep 15, 2008 at 17:00

Lucas Oman

15.9k ● 2 ● 46 ● 45

Google Analytics's code detects HTTPS on its own and serves from a secure server. They've had this for over a year now, and users using the older code can upgrade whenever they feel like. – ceejayoz Sep 15, 2008 at 17:28

Hey, thanks for that. I didn't know. I guess I should read their blog a little more often ;-) – Lucas Oman Sep 15, 2008 at 18:22

Doesn't it defeat the purpose of secure pages if you're going to let content in from third party sources. Espcially in the case of Javascript files (like Google analytics) that could potentially act as keyloggers and send secure information to the third party? – Kibbee Jan 11, 2009 at 18:29

1

I would say the most common mistakes when working with an SSL-enabled site are

1. The site erroneously redirects users to http from a page as https
2. The site doesn't automatically switch to https when it's necessary
3. Images and other assets on an https page are being loading via http, which will trigger a security alert from the browser. Make sure all assets are using fully-qualified URIs that specify https.
4. The security certificate only works for one subdomain (such as www) but your site actually uses multiple subdomains. Make sure to get a wildcard certificate if you will need it.

Share Improve this answer

Follow

answered Sep 15, 2008 at 17:02

Peter Bailey

106k ● 32 ● 185 ● 206

0

I would suggest any time *any* user data is stored in a database and communicated, use https. Consider this requirement even if the user data is mundane, because even many of these mundane details are used by that user to identify themselves on other websites. Consider all the random security questions your bank asks you (like what street do you live on?). This can be taken from address fields really easily. In this case, the data is not

what you consider a password, but it might as well be. Furthermore, you can never anticipate what user data will be used for a security question elsewhere. You can also expect that with the intelligence of the average web user (think your grandmother) that that tidbit of information might make up part of that user's password somewhere else.

One pointer if you use https

make it so that if the user types http://www.website-that-needs-https.com/etc/yadda.php they will automatically get redirected to https://www.website-that-needs-https.com/etc/yadda.php (personal pet peeve)

However, if you're just doing a plain html webpage, that will be essentially a one-way transmission of information from the server to the user, don't worry about it.

Share Improve this answer

Follow

answered Sep 15, 2008 at 17:01

Doug T.

65.5k ● 28 ● 141 ● 205

0

All very good tip here... but I just want to add something.. Ive seen some sites that gives you a http login page and only redirect you to https after you post your username/pass.. This means the username is transmitted in the clear before the https connection is established..

In short make the page where you login from ssl, instead of posting to an ssl page.

Share Improve this answer

Follow

answered Sep 15, 2008 at 22:51

paan

7,182 ● 8 ● 40 ● 44

0

I found that trying to `<link>` to a non-existent style sheet also caused security warnings. When I used the correct path, the lock icon appeared.

Share Improve this answer

Follow

answered Oct 16, 2008 at 20:56

JSchaefer

7,261 ● 4 ● 21 ● 10