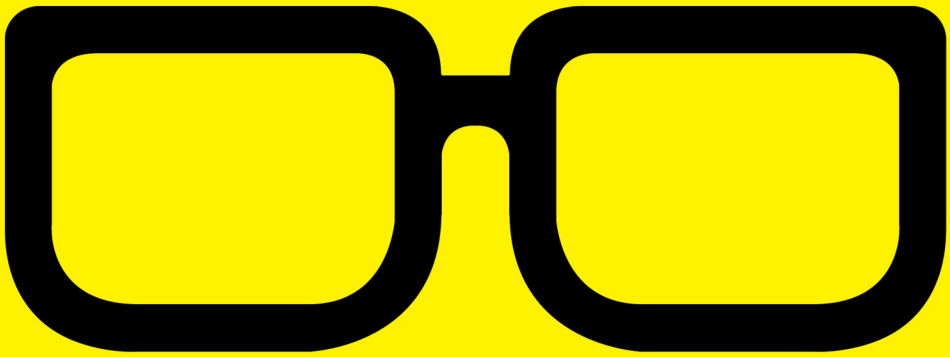
SPONSORED BY

GeoTrust®

GEEK GUIDE



Apache Web Servers and SSL Authentication

LINUX™ JOURNAL

Table of Contents

REUVEN M. LERNER is a Web developer, consultant, trainer and longtime columnist for *Linux Journal*. He recently completed his PhD in Learning Sciences from Northwestern University. You can read his blog, Twitter feed and newsletter at http://lerner.co.il. Reuven lives with his wife and three children in Modi'in, Israel.

GEEK GUIDES:

Mission-critical information for the most technical people on the planet.

About the Sponsor

GeoTrust—A Trusted Leader in Online Security Services

GeoTrust is the world's second largest digital certificate provider. More than 100,000 customers in over 150 countries trust GeoTrust to secure online transactions and conduct business over the Internet. Our range of digital certificate and trust products enable organizations of all sizes to maximize the security of their digital transactions cost-effectively.

GeoTrust's world-class SSL Certificates (https://www.geotrust.com/ssl/) offer fast delivery at a cost-effective price, enabling up to 256-bit SSL encryption, and include a range of GeoTrust True Site Seals based on the level of identity verification.

GeoTrust's Signing Products (https://www.geotrust.com/signing-products) represent the latest in next-generation technology for digitally signing applications, binding people and documents, and assuring code integrity to wireless platforms.

The GeoTrust GeoCenter, a robust management portal, offers 24/7 control, instant issuance, and volume pricing for GeoTrust Partners (https://www.geotrust.com/sell-ssl-certificates/#) and Enterprise SSL (https://www.geotrust.com/enterprise-ssl-certificates/enterprise-ssl/) customers.

Apache Web Servers and SSL Authentication

REUVEN M. LERNER

Overview

Congratulations! You've decided to set up a Web site. The site might be for your personal use, for sharing family pictures, for a blog, for an SaaS application, or any number of other possibilities. In all of those cases, people will access your site using the Hypertext Transfer Protocol (HTTP). HTTP has evolved and improved through the years, but one thing about it hasn't changed—the fact that all of the traffic sent

on an HTTP connection is unencrypted.

If you're running a personal site, this doesn't really matter. But if and when you're interested in doing something a bit more exciting, you might well want or need to add encryption to your server. Encrypted HTTP is known as HTTPS, and it involves the addition of SSL/TLS encryption to the HTTP protocol. In order for your site to handle encrypted connections, you need to install and configure an SSL certificate. This Geek Guide is here to help you through that process, making it easy for you to move an existing site using HTTP into one using HTTPS.

There are several motivations for using HTTPS, rather than HTTP, on Web sites. One is that a growing number of sites are engaging in commerce, and many payment providers (and their security standards, known as PCI) are requiring that commercial sites not only encrypt traffic having to do with payments, but also with logins and other aspects.

In some cases, sites are either required to keep information as private as possible, or they wish to do so in order to avoid regulatory problems. Medical and legal offices, for example, would want to avoid having others eavesdrop on discussions employees are having and e-mail messages they are sending and receiving. Without a secure channel, it might be possible for someone to listen in, without actually gaining access to the network.

Some users may be more likely to trust a site that is encrypting traffic. Thus, some sites use HTTPS not because they need to do so, but because they know users might feel more comfortable with an HTTPS connection in place.

Gone are the days when running an HTTPS server was difficult, time-consuming or too CPU-intensive for a server; today's computers are more than powerful enough to handle HTTPS traffic.

Finally, some people say that given how inexpensive and easy it is to use SSL, and to use an encrypted version of HTTP, that they might as well do so. Gone are the days when running an HTTPS server was difficult, time-consuming or too CPU-intensive for a server; today's computers are more than powerful enough to handle HTTPS traffic.

The bottom line is that whether you want or need to do so, adding HTTPS to a site you're running isn't very hard to do. In this Geek Guide, I walk through what SSL/TLS is (and isn't), how you can create or buy a certificate, how to install that certificate into an Apache server and then how to configure Apache such that a subset of URLs on your system are covered by SSL.

Introduction to SSL/TLS

First, it's important to understand some terminology and technology. HTTPS differs from plain-old HTTP in that it is "secure". However, people often are surprised to discover that this means the *connection* between the users' browsers

and the server is secure. It doesn't guarantee anything about the security of the server or of their data. All that HTTPS ensures is that if you're using a café's Wi-Fi to purchase something on Amazon, no one will be able to read your user name and password as they are sent to Amazon's login system.

The security in HTTPS uses a protocol known as transport layer security, or TLS for short. The current stable version of TLS is 1.2, with version 1.3 currently in draft form. Although TLS has been around since 1999, and replaced the earlier SSL (secure socket layer) protocol invented by Netscape, many people continue to refer to the protocol as SSL, even though currently no version of SSL is considered to be secure. Thus, when people mention an SSL certificate, they almost certainly actually are referring to a TLS certificate. The naming difference, it would seem, had little to do with substantial changes to the protocols and was more political than technical in nature.

It doesn't help that the best-known open-source library for working with TLS is still known as OpenSSL. You can read more about the OpenSSL library at its Web site: http://openssl.org.

TLS (and SSL) use public-key cryptography to ensure that communication between a user's browser and a Web server is kept secure. In public-key cryptography, each party has two keys, one public and one private. The public key (as its name implies) can be shared freely with the world, while the private key (as its name implies) must be kept secret.

The two keys are inverses of each other, allowing you to use

them in two different ways. You can encrypt text and ensure that only a specific person can read it, using that person's public key. Only by using the private key can the user then decrypt the information. You also can use the keys to sign documents digitally. Encrypt a document using your private key, and everyone can decrypt the document, but they also will know that you, and only you, could have signed it.

You can imagine, given this description, that you can use these keys to ensure that traffic between a browser and server are encrypted and, thus, unreadable to anyone without the appropriate keys. A site could, in theory, distribute its public key to anyone and everyone. A browser then could use the public key to encrypt HTTP traffic. Only the site would be able to read that traffic, because only it would have the private key.

However, things in the TLS world are a bit more complex than that. For starters, remember that you need to have bidirectional encryption, handling not only requests sent to the server, but also responses sent to the browser. Add to that fact the logistical aspect of distributing a server's public keys in order for a browser to connect with HTTPS. And then you also have the issue of ensuring that the key you receive is a genuine one, and not the result of a "man in the middle" attack.

The bidirectional part of TLS is perhaps easiest to explain: the actual public and private keys aren't directly used to encrypt the traffic. Rather, they're used to create a "session key" (that is, an encryption key that is used for a single client-server session), and to exchange it securely.

Distributing the server's keys in a reasonable way is a bit trickier.

Distributing the server's keys in a reasonable way is a bit trickier. After all, if I go to a server, how does my browser know if the server's public key (distributed in a document known as a "certificate", along with some other data) is genuine or the result of someone having broken into the server?

The answer is that the server's public key is digitally signed, as described above, not by the server itself, but from an organization known as a certificate authority, or CA. Your browser comes preconfigured with the public keys of a large number of CAs. Thus, when a server sends its public key signed by a well known CA, your browser can guarantee the truth of the signature and then use the server's public key as part of its creation of a shared session key between the browser and server.

Making your Web site secure is thus a matter of creating a set of certificates (public and private) for your server, signed by a well known CA. This Geek Guide walks through the process of doing just that.

Types of Certificates

A certificate contains not only the server's key, but also some information about the server to ensure that the key is legitimate. For example, if I create a certificate

for XYZ Corp., I cannot use it on a server run by JKL Corp. The certificate is issued for a particular hostname, providing an even greater degree of security and trust.

If you're going to be securing a single site on a single domain, you probably can get this simplest type of TLS certificate. When you create the certificate, you must describe the location, the organization and the full hostname of the server in question. Note that the certificate is based on the hostname and not on the IP address; this means if you move your server to a new location (but keep the same name), you likely won't need to change the certificate, so long as the hostname remains the same.

However, a growing number of sites cannot use such simple, single-hostname certificates. The simplest example of such a case is a Web site with multiple hostnames, such as "example.com" and "www.example.com". In such a case, you would have to purchase and configure two separate certificates, one for each hostname. Alternatively, you could buy and configure what's known as a wildcard certificate, which allows you to use a single certificate with a number of hosts within a particular domain. The more hosts you want to secure under a specific domain, the more a wildcard certificate will save you time and money.

It's important to understand that wildcard certificates are good only for a single domain. Thus, if your company runs three different on-line stores, each of which with its own domain, you will need to purchase three separate certificates. You cannot use a wildcard in that case. In such a situation, you'll need to indicate the domain (but not the

If you are running three (or more) domains on your server, and particularly if you think you might need to add more domains to the same server, you should consider a unified communications certificate, otherwise known as a UCC.

hostnames) that should be associated with the certificate at purchase time. The hostname cannot be changed once the certificate is issued, which means if you aren't completely sure about your domain name, it might be wise to wait a bit before making the purchase.

If you are running three (or more) domains on your server, and particularly if you think you might need to add more domains to the same server, you should consider a unified communications certificate, otherwise known as a UCC. The name might sound complex, but it isn't. The basic idea behind a UCC is that you can add and remove domains from the certificate during its lifetime. You don't need to know the domain names when the certificate is created; you can, after the certificate has been issued, modify and re-issue it with the latest hostnames. Each change to the certificate requires re-installing it on your server, but as you'll see, the installation process is quite straightforward and easy. Each alternative domain name on a certificate is known as a subject alternative name (SAN), and the number of SANs is the main hard-coded limit on a UCC.

For example, one of my clients needed a TLS certificate for a site. However, the site used two different domain names, and we knew when we ordered the certificate that we would be adding about ten additional domains within a year—with an eye toward about 60 total. So, we bought a UCC that allowed for up to 100 SANs. Every time we registered a new domain name for the site, we would re-generate the certificate to include the new name, and then install this new version. However, the UCC allowed us to add and remove SANs without incurring additional costs or the administrative overhead of working with many different certificates.

Certificate Authorities

Once you have decided what kind of certificate you want to install, you need to consider the certificate authority (CA) you want to use. At the end of the day, a CA is there to guarantee that the certificates it issues are valid, and that the information the certificates contain hasn't been tampered with. As a result, you want to choose a CA whose certificates have been installed in a large number of browsers. As a general rule, nearly any CA vendor will be recognized by your browser, but if you're going to purchase from a smaller operation, it's probably best to be sure, just in case. It's true that the CA's name is mentioned in the SSL certificate. Thus, if you go to a Web site and click on the HTTPS logo, you should be given an option to see the name of the CA that authorized the certificate.

If you use a certificate whose CA isn't recognized by all browsers, some users will be presented with a (rather scary) warning, telling them not to trust this site's certificate.

If you use a certificate whose CA isn't recognized by all browsers, some users will be presented with a (rather scary) warning, telling them not to trust this site's certificate. In most cases, I have found that such warnings are because of innocent misconfiguration of the server. However, I can almost guarantee that your users will not accept a certificate that produces such warnings.

However, there are cases in which you don't want or need to get a certificate from a CA. For example, if you just want to set up SSL for an experiment or for your own development machine, you almost certainly don't want to pay money just to get a certificate issued. In such cases, you can self-sign your certificate.

I should stress that there is no technical barrier to always self-signing certificates. You could, in theory, create a multi-billion dollar e-commerce empire using only self-signed certificates. However, the odds of being able to do this are slim, because so many modern browsers will warn users—often with bold, red warnings that are quite scary to the uninitiated—that

they are visiting a site whose certificate is signed by a CA unknown to the browser. Might the TLS certificate be legitimate? Yes, but there's no way to know that for sure, because the browser doesn't have any way to validate the digital signature on the server's certificate.

So although it's easy and fast to self-sign a certificate, you should do so only when you are sure that the general public won't be relying on that certificate for anything that directly affects your business.

Getting Ready for SSL/TLS

This Geek Guide describes how to install an SSL certificate on your Apache HTTP server. Of course, other HTTP servers exist. In the past few years, nginx has become particularly powerful. Apache is not only popular, but it's also quite flexible. Indeed, its claim to fame is that Apache is based on a large number of modules, any of which can be included or excluded in your server configuration. This not only allows you to create what's effectively a custom HTTP server for your purposes, but it also (if you're interested in doing so) offers the option of writing your own custom modules.

The Apache module that handles SSL/TLS is known as mod_ssl and is included in the basic Apache configuration. In order to have a secure server, not only must you have mod_ssl installed, but it also must be activated. On many modern Linux distributions, such as Ubuntu, the Apache configuration directory has a mods-available subdirectory, as well as a mods-installed

subdirectory. Typically, modules are placed inside mods-available; if you actually want to use a module, you create a symlink from mods-installed back to mods-available. On one of my Apache servers, I have about 30 modules in mods-available, but only about 15 links back to them in mods-installed.

Installing a module means adding two symlinks: one to a .conf configuration file containing the modules' configuration, as well as a .load file, which typically contains a single Apache `LoadModule` directive, telling Apache where the compiled module is located.

Now, you could create these symlinks yourself. But as a general rule, it's easier to let Apache take care of this for you, using the alias `a2enmod` command that comes with many modern Debian-based Linux distros. Thus, to enable the `mod_ssl` module on Ubuntu or Debian, you simply can run:

```
$ sudo e2enmod ssl
```

Now that the SSL module is installed, you'll need to order or create an SSL certificate. I'm going to assume here that you'll be going to your favorite CA and ordering a certificate. However, before you take out your wallet and go to the CAs Web site, you'll first need to do some work on your server. That's because a CA can create a certificate only if it first has received a "certificate signing request", known as a CSR. Funny

Given that the entire TLS infrastructure is based on public and private keys, it won't surprise you to hear that you'll need to generate a pair of keys and then use those keys to create your CSR.

as it might sound, this basically means creating a certificate (the CSR) that you give to the CA. The CA uses the CSR to create your SSL certificate. You then download the actual certificate and install it on your server.

Given that the entire TLS infrastructure is based on public and private keys, it won't surprise you to hear that you'll need to generate a pair of keys and then use those keys to create your CSR. You'll do this using the `openssl` command, which comes with the OpenSSL library. If you don't see the `openssl` command on your computer, you'll need to install OpenSSL from your distribution's archive.

If this is the first time you're dealing with any of this, you'll need to generate a new keypair, as well as the CSR itself. Here is the (admittedly long) command you'll need to run in order to get this to work:

```
$ sudo openssl req -new -newkey rsa:2048
➥-nodes -keyout key.pem -out req.pem
```

Let's take apart the above command:

- `openssl req -new` tells the OpenSSL command that you're creating a new CSR.
- The `-newkey` option indicates that you want to generate a new keypair, rather than use an existing one.
- The `rsa:2048` tells the "newkey" option that you want to generate a 2,048-bit RSA keypair.
- The `-nodes` option indicates that you don't want to encrypt the private key with a passphrase. On the one hand, this is indeed less secure and means that someone can restart your Apache server without knowing the passphrase. On the other hand, do you want to have to enter the passphrase every time you restart your server? More security-minded folks than myself, or people with staff members on duty 24/7 who can take care of restarting servers, might want to remove this option.
- The `-keyout key.pem` option tells OpenSSL where it should store the newly created private key. The output is in PEM format, which is frequently used with OpenSSL, which explains the file extension. You can use whatever filename you want, of course.
- Finally, "-out req.pem" tells OpenSSL where it should store the CSR that it creates.

Once you run the above command, you'll be prompted to enter a bunch of details about your server—where it's located, the company name and your name and e-mail address.

At the end of this process, OpenSSL will create the two files that you requested: key.pem and req.pem.

At this point, you have a CSR. You can use it to request a TLS certificate from your favorite CA, or you can use it to self-sign a TLS certificate created on your own machine. If you use a commercial CA, you'll submit the CSR and then you'll probably have to wait a bit for it to be processed and approved, and for you to get the certificate. You'll likely receive e-mail from your CA, telling you that the certificate is now available for download.

Installing the Certificate

Although people might make a big deal out of TLS certificates, the fact is that they're just files—digitally signed files maybe, but files nonetheless. Thus, when you download the certificate from your CA's Web site, you'll have a file that you then must put on your server. Actually, you'll likely have to put three files on your server, one of which you already have:

- The key.pem file, containing the private key from your server's keypair.
- The TLS certificate that you received from your CA, which likely will have a .crt extension.

- The "Chain file" from your CA, which is used as part of the identification process with your CA. This also likely will have a .crt extension.

On Debian-based systems, these files typically are placed in /etc/apache2/ssl. I generally put the private key (that is, the key.pem file) under the "private" subdirectory and the .crt files in the "certs" subdirectory.

You now have installed your certificate! Moreover, you have all of the pieces you need in order for it to work. But, you still haven't told Apache to activate SSL or to use your certificates.

In order to do that, you'll need to return to your Apache configuration. Whereas HTTP typically uses port 80, HTTPS generally uses port 443. Thus, you'll probably want to set up a separate virtual host on port 443, in which you'll set up your HTTPS configuration. The config can be as simple as the following:

```
<VirtualHost *:443>
  ServerName mydomain.com
  DocumentRoot /var/www/www.mydomain.com/

  SSLEngine on
  SSLOptions +FakeBasicAuth +ExportCertData +StrictRequire

  SSLCertificateFile /etc/apache2/ssl/certs/mydomain.com.crt
  SSLCertificateKeyFile /etc/apache2/ssl/private/mydomain.key
  SSLCertificateChainFile /etc/apache2/ssl/certs/myca.crt
</VirtualHost>
```

The above configuration would be for the server at MyDomain.com; it's probably safe to assume that you're working on a server at a different domain. The file starts by opening a <VirtualHost> section, indicating to Apache that you want to configure this virtual host for requests to mydomain.com on port 443. Note that if you don't set `ServerName`, you will get errors from Apache.

If you have successfully added the `ssl` module to your Apache configuration, as described above, the SSL* configuration directives should be available and should work. Using the directives without installing the module (and then restarting Apache) will result in error messages, indicating that you are trying to use undefined directives.

The above configuration uses a number of different SSL-related directives and options:

- `SSLEngine on`, as you can imagine, turns on the SSL system. SSL can be a heavy burden on some servers (although not nearly as much as it used to be), and including the module isn't enough to get SSL features working.

- As is the case with many Apache modules, `mod_ssl` offers a number of options. In the above configuration, I have activated three: `FakeBasicAuth` (which allows you to use standard Apache authentication for user access), `ExportCertData` (which passes along several environment variables having to do with SSL to the Web

It's common to want to redirect traffic from your HTTP site to your HTTPS site.

application) and `StrictRequire` (which allows you to force SSL on particular pages).

You then tell `mod_ssl` where it can find the three files it needs in order to run SSL successfully:

- `SSLCertificateFile` points to the certificate that you received from your CA.
- `SSLCertificateKeyFile` points to the private key that you generated earlier with the `openssl` command.
- `SSLCertificateChainFile` points to a certificate that you should have received from your CA.

With all of the above in place, you should be able to restart your Apache server! Then, assuming that your site is at MyDomain.com, you can go to https://MyDomain.com/. If all is configured correctly, then this should answer. However, now you have a bit of an issue—your site likely is responding on both HTTP and HTTPS, and might well be configured differently. It's common to want to redirect traffic from your HTTP site to your HTTPS site. In other

words, if I go to http://MyDomain.com/, the site should redirect me to https://MyDomain.com/. One simple way to do that is to use the `Redirect` directive to map an old URL to a new one. For example, on your HTTP (non-TLS) site, you could say:

```
Redirect permanent / https://MyDomain.com/
```

This will work only for your home page; if you have a large number of URLs that you want to redirect, it's not sufficient. In such a case, you might well need to use `mod_rewrite`, which comes with Apache and allows you to use regular expressions to grab parts of URLs and redirect people to there. The advantage of the `Redirect` directive is that it's simple to create, and it allows you to pick and choose the URLs that will be redirected to HTTPS.

One of the issues that often surprises people when working with SSL is that it's a bit harder to debug than regular HTTP communication, because everything is encrypted. One of my favorite techniques to see if a site is working is to telnet to port 80 and issue a `GET / HTTP/1.0` command by hand. Of course, there's no way to do that with HTTPS! Instead, you might well need to use curl, wget or even your browser to make requests to your server, and then turn up the logging level on Apache to make sure that everything you're doing is logged and can be analyzed. You also can set, in your Apache configuration (on my system,

in mods-available/ssl.conf), the debugging of SSL to a higher level:

```
<IfModule mod_ssl.c>
    ErrorLog /var/log/apache2/ssl_engine.log
    LogLevel debug
</IfModule>
```

This will ensure that whatever happens in mod_ssl is logged, and thus can be debugged more easily. Note that the "debug" log level is rather verbose; you likely will have to wade through a fair amount of text to find what you want. However, that's better than not being able to find anything useful, which is what the "info" level often ends up doing.

Another problem that people sometimes have is a mismatch between the server name and the name on the certificate. When you create the CSR, you must use the right hostname. You can't just move it around willy-nilly. And, if you change the name of your server or your domain, the SSL certificate no longer will be valid.

Conclusion

SSL (and TLS) used to be big and complex, and difficult for people to install. Today, making your site secure is often a necessary part of doing business—either because you must do it or because you want your customers to trust you. If you use Apache and Linux, configuring your server to use TLS isn't that difficult. The tools are fairly easy to understand and use, and they come with every installation of Apache.■

Resources

Perhaps the best guide to Apache and SSL is, not surprisingly, the documentation on Apache's own site. The main page on the subject is at https://httpd.apache.org/docs/2.2/ssl.

If you're looking for the specific SSL-related directives, you will need the mod_ssl documentation: https://httpd.apache.org/docs/2.2/mod/mod_ssl.html.