CHAPTER 7
SSL

Secure Socket Layers (SSL) is the standard way to implement secure Web sites. By encrypting the traffic between the server and the client, which is what SSL does, that content is protected from a third party listening to the traffic going past.

All of the traffic exchanged is encrypted once the SSL session has been set up. This means that even the URLs being requested are encrypted.

The exact mechanism by which this encryption is accomplished is discussed extensively in the SSL specification, which you can read at *http://wp.netscape.com/eng/ssl3/*. For a more user-friendly discussion of SSL, we recommend looking through the *mod_ssl* manual, which you can find at *http://httpd.apache.org/docs/2.2/ssl*. This document discusses not only the specific details of setting up *mod_ssl* but also covers the general theory behind SSL it and has pictures illustrating the concepts.

You also may wish to see the TLS 1.0 (RFC 2246) specification, which provides what might be thought of as the next generation of SSL. You can read the full specification at *http://www.ietf.org/rfc/rfc2246.txt*, or a more friendly explanation at *http://en.wikipedia.org/wiki/Transport_Layer_Security*.

In this chapter, we talk about some of the common things that you might want to do with your secure server, including how to install it.

7.1 Installing SSL

Problem

You want to install SSL on your Apache server.

Solution

The solutions to this problem fall into several categories, depending on how you installed Apache in the first place (or whether you are willing to rebuild Apache to get SSL).

If you installed a binary distribution of Apache, your best bet is to return to the place from which you acquired that binary distribution, and try to find the necessary files for adding SSL to it.

If you built Apache yourself from source, then the solution will depend on whether you are running Apache 1.3 or Apache 2.x.

In Apache 1.3, SSL is an add-on module, which you must acquire and install from a different location than that from where you obtained Apache. There are two main choices available: *mod_ssl* (*http://www.modssl.org/*) and Apache-SSL (*http://www.apache-ssl.org/*); the installation procedure will vary somewhat depending on which one of these you choose.

If you are building Apache 2.x from source, the situation is somewhat simpler; just add *--enable-ssl* to the *./configure* arguments when you build Apache to include SSL as one of the built-in modules.

Consult Chapter 1 and Chapter 2 for more information on installing third-party modules, particularly if you have installed a binary distribution of Apache rather than building it yourself from the source code.

If you are attempting to install SSL on Apache for Windows, there is a discussion of this in the Compiling on Windows document, which you can find at *http://httpd.apache.org/docs/2.0/platform/win_compiling.html* for Apache 2.0. Or if you are using Apache 1.3 on Windows and wish to install SSL, you should consult the file *INSTALL.Win32*, which comes with the SSL distribution, or look at the HowTo at *http://tud.at/programm/apache-ssl-win32-howto.php3*.

Finally, note that the Apache SSL modules are an interface between Apache and the OpenSSL libraries, which you must install before any of this can work. You can obtain the OpenSSL libraries from *http://www.openssl.org/*. Although you may already have these libraries installed on your server, it is recommended that you obtain the latest version of the libraries to have the most recent security patches and to protect yourself from exploits.

Discussion

So, why is this so complicated? Well, there are a variety of reasons, most of which revolve around the legality of encryption. For a long time, encryption has been a restricted technology in the United States. Because Apache is primarily based out of the United States, there is a great deal of caution regarding distributing encryption technology with the package. Even though major changes have been made in the laws, permitting SSL to be shipped with Apache 2.0, there are still some gray areas that make it problematic to ship compiled binary distributions of Apache with SSL enabled.

This makes the situation particularly unpleasant on Microsoft Windows, where most people do not have a compiler readily available to them, and so must attempt to acquire binary builds from third parties to enable SSL on their Apache server on Windows. The

URL given previously for compiling Apache 2.0 with SSL on Windows assumes that you do have a compiler, and the document telling you how to build Apache 1.3 with SSL takes great pains to encourage you not to use Apache 1.3 on Windows, where it does not have comparable performance to Apache on Unixish operating systems.

See Also

- *http://httpd.apache.org/docs-2.0/platform/win_compiling.html*
- *http://tud.at/programm/apache-ssl-win32-howto.php3*
- *http://www.openssl.org/*
- *http://www.modssl.org/*
- *http://www.apache-ssl.org/*

7.2 Installing SSL on Windows

Problem

You want to install Apache with SSL on Microsoft Windows

Solution

Obtain XAMPP from *http://apachefriends.org/* and install that.

Discussion

As was mentioned in the previous recipe, it is certainly possible to build Apache with SSL from source on Microsoft Windows. However, to be honest, this is beyond the expertise of most of us.

Fortunately, the kind folks at ApacheFriends have made available a binary distribution called XAMPP which includes, among other things, Apache with *mod_ssl*. The package also includes MySQL, PHP and Perl, some of the commonly used tools in web site development.

So, save yourself some pain, take advantage of the great work that has been done by the ApacheFriends guys, and install the XAMPP package.

7.3 Generating Self-Signed SSL Certificates

Problem

You want to generate a self-signed certificate to use on your SSL server.

Solution

Use the *openssl* command-line program that comes with OpenSSL:

```
% openssl genrsa -out server.key 1024
% openssl req -new -key server.key -out server.csr
% cp server.key server.key.org
% openssl rsa -in server.key.org -out server.key
% openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Then move these files to your Apache server's configuration directory, such as */www/conf/*, and then add the following lines in your *httpd.conf* configuration file:

```
SSLCertificateFile "/www/conf/server.crt"
SSLCertificateKeyFile "/www/conf/server.key"
```

Discussion

The SSL certificate is a central part of the SSL conversation and is required before you can run a secure server. Thus, generating the certificate is a necessary first step to configuring your secure server.

Generating the key is a multistep process, but it is fairly simple.

Generating the private key

In the first step, we generate the private key. SSL is a private/public key encryption system, with the private key residing on the server and the public key going out with each connection to the server and encrypting data sent back to the server.

The first argument passed to the *openssl* program tells *openssl* that we want to generate an RSA key (*genrsa*), which is an encryption algorithm that all major browsers support.

You may, if you wish, specify an argument telling *openssl* something to use as the source of randomness. The **-rand** flag will accept one or more filenames, which will be used as a key for the random number generator. If no **-rand** argument is provided, OpenSSL will attempt to use */dev/urandom* by default if that exists, and it will try */dev/random* if */dev/urandom* does not exist. It is important to have a good source of randomness in order for the encryption to be secure. If your system has neither */dev/urandom* nor */dev/random*, you should consider installing a random number generator, such as *egd*. You can find out more information about this on the OpenSSL Web site at *http://www.openssl.org/docs/crypto/RAND_egd.html*.

The **-out** argument specifies the name of the key file that we will generate. This file will be created in the directory in which you are running the command, unless you provide a full path for this argument. Naming the key file after the hostname on which it will be used will help you keep track of the file, although the name of the file is not actually important.

And, finally, an argument of 1024 is specified, which tells *openssl* how many bytes of randomness to use in generating the key.

You should see some output something like:

```
Generating RSA private key, 1024 bit long modulus
.....................................................++++++
........++++++
e is 65537 (0x10001)
```

Generating the certificate signing request

The next step of the process is to generate a certificate signing request. The reason it is called this is because the resultant file is usually sent to a *certificate authority* (CA) for signing and is, therefore, a signing request. (A certificate is just a signed key, showing that someone certifies it to be valid and owned by the right entity.)

A certificate authority is some entity that can sign SSL certificates. What this usually means is that it is one of the few dozen companies whose business it is to sign SSL certificates for use on SSL servers. When a certificate is signed by one of these certificate authorities browsers will automatically accept the certificate as being valid. If a certificate is signed by a CA that is not listed in the browser's list of trusted CAs, then the browser will generate a warning, telling you that the certificate was signed by an unknown CA and asking you if you are sure that you want to accept the certificate.

This is a bit of an oversimplification of the process but conveys enough of it for the purposes of this recipe.

The alternative is that you sign the certificate yourself, which is what we'll be doing in the coming steps.

The arguments to this command specify the key for which the certificate is being generated (the **-key** argument) and the name of the file that you wish to generate (the **-out** argument).

If you want a certificate that will be accepted without warning or comment by all major browsers, you will send the *csr* file, along with a check or credit card information, to one of these CAs.

During this step, you'll be asked a number of questions. The answers to these questions will become part of the certificate, and will be used by the browser to verify that the certificate is coming from a trusted source. The end user may inspect these details any time they connect to your Web site.

The questions will look like the following:

```
Country Name (2 letter code) [GB]: EX
State or Province Name (full name) [Berkshire]: CO
Locality Name (eg, city) [Newbury]: Example City
Organization Name (eg, company) [My Company Ltd]: Institute of Examples
Organizational Unit Name (eg, section) []: Demonstration Services
```

```
Common Name (eg, your name or your server's hostname) []: www.example.com
Email Address []: big-cheese@example.com
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

All of these values are optional, with the exception of the Common Name. You must supply the correct value here, which is the hostname of the server on which this certicate will be used. It is crucial that the hostname that you put in here exactly match the hostname that will be used to access the site. Failure to do this will result in a warning message each time a user connects to your Web site.

Removing the passphrase

In the first step, we put a passphrase on the private key. This makes the key encrypted, so that only someone with the passphrase can read the contents of the key.

A side-effect of this is that every time you start up your Apache server, you will need to type in the passprase. This is extremely inconvenient, as it means that starting up the Web server always requires a manual step. This is particularly a problem for reboots, or other automated restarts of the Apache server, when there might not be a human handy to type in the passphrase.

Therefore we're going to remove the passphrase from the key, so that this isn't an issue.

The key is copied to a backup location just in case we screw something up, and then the command is issued to remove the passphrase, resulting in an unencrupted key. You must remember to change the permissions on the file so that only root can read this file. Failure to do so may result in someone stealing that file and then being able to run a Web site pretending to be you.

Signing your key

If you choose not to send the CSR to a Certificate Authority, and, instead, sign your own public key (also called "signing your own certificate," because signing your public key results in a self-signed certificate), this will result in a perfectly usable certificate, and save you a little money. This is especially useful for testing purposes, but it may also be sufficient if you are running SSL on a small site or a server on your internal network.

The process of signing a key means that the signer trusts that the key does indeed belong to the person listed as the owner. If you pay Entrust or one of the other commercial CAs for a certificate, they will actually do research on you and verify, to some degree of certainty, that you really are who you claim to be. They will then sign your public key and send you the resulting certificate, putting their stamp of approval on it and verifying to the world that you are legitimate.

In the example given, we sign the key with the key itself, which is a little silly, as it basically means that we trust ourselves. However, for the purposes of the actual SSL encryption, this is sufficient.

If you prefer, you can use the *CA.pl* script that comes with OpenSSL to generate a CA certificate of your own. The advantage of this approach is that you can distribute this CA certificate to users, who can install it in their browsers, enabling them to automatically trust this certificate and any other certificates that you create with that same CA. This is particularly useful for large companies where you might have several SSL servers using certificates signed by the same CA.

Of the arguments listed in the command, one of the most important ones is the `-days` argument, which specifies how many days the certificate will be good for. If you are planning to purchase a commercial certificate, you should generate your own self-signed key that is good for perhaps 30 days, so that you can use it while you are waiting for the commercial certificate to arrive. If you are generating a key for actual use on your server, you may want to make this a year or so, so that you don't have to generate new keys very often.

The `-signkey` argument specifies what key will be used to sign the certificate. This can be either the private key that you generated in the first step or a CA private key generated with the *CA.pl* script, as mentioned earlier.

If this step goes well, you should see some output like the following:

```
Signature ok
subject=/C=US/ST=KY/L=Wilmore/O=Asbury College/OU=Information
Services/CN=www.asbury.edu/Email=rbowen@asbury.edu
Getting Private key
```

Configuring the server

Once you have generated the key and certificate, you can use them on your server using the two lines of configuration shown in the previous solution.

The easy way

Now that we've gone through the long and painful way of doing this, you should know that there is a simpler. OpenSSL comes with a handy script, called *CA.pl*, which simplifies the process of creating keys. The use of *CA.pl* is described in Recipe 7.4 so you can see it in action. It is useful, however, to know some of what is going on behind the script. At least, we tend to think so. It also gives you considerably more control as to how the certificate is made.

See Also

- The *man* page for the *openssl* tool
- The *man* page for the *CA.pl* script

- *CA.pl* documentation, at *http://www.openssl.org/docs/apps/CA.pl.html*

7.4 Generating a Trusted CA

Problem

You want to generate SSL keys that browsers will accept without a warning message.

Solution

Issue the following commands:

```
% CA.pl -newca
% CA.pl -newreq
% CA.pl -signreq
% CA.pl -pkcs12
```

Discussion

Recipe 7.3 discusses the lengthy steps that are required to create keys and sign them. Fortunately, OpenSSL comes with a script to automate much of this process, so that you don't have to remember all of those arguments. This script, called *CA.pl*, is located where your SSL libraries are installed, for example, */usr/share/ssl/misc/CA.pl*.

The lines in the Solution hide a certain amount of detail, as you will be asked a number of questions in the process of creating the key and the certificate. Note also that you will probably need to be in the directory where this script lives to get successful results from this recipe.

If you want to omit the passphrase on the certificate so that you don't have to provide the passphrase each time you start up the server, use *-newreq-nodes* rather than *-newreq* when generating the certificate request.

After running this sequence of commands, you can generate more certificates by repeating the *-newreq* and *-signreq* commands.

Having run these commands, you will have generated a number of files. The file *newcert.pem* is the file you specify in your *SSLCertificateFile* directive, the file *newreq.pem* is your *SSLCertificateKeyFile*, and the file *demoCA/cacert.pem* is the CA certificate file, which will need to be imported into your users' browsers (for some browsers) so that they can automatically trust certificates signed by this CA. And, finally, *newcert.p12* serves the same purpose as *demoCA/cacert.pem* for certain other browsers.

Importing the CA

If your users are using Internet Explorer, you need to create a special file for them to import. Use the following command:

```
openssl X509 -demoCA/cacert.pem -out cacert.crt -outform DER
```

Then you can send them the *cacert.crt* file.

Clicking on that file will launch the SSL certificate wizard and guide the user through installing the CA certificate into their browser.

Other browsers, such as Mozilla, expect to directly import the *cacert.pem* file. Users will navigate through their menus (Edit => Preferences => Privacy and Security => Certificates), then click on Manage Certificates, then on the Authorities tab, and finally on Import, to select the certificate file.

After importing a CA certificate, all certificates signed by that CA should be usable in your browser without receiving any kind of warning.

See Also

- The manpage for the *CA.pl* script
- *CA.pl* documentation at *http://www.openssl.org/docs/apps/CA.pl.html*

7.5 Serving a Portion of Your Site via SSL

Problem

You want to have a certain portion of your site available *via* SSL exclusively.

Solution

This is done by making changes to your *httpd.conf* file.

For Apache 1.3, add a line such as the following:

```
Redirect /secure/ https://secure.example.com/secure/
```

For Apache 2.0:

```
<Directory /www/secure>
    SSLRequireSSL
</Directory>
```

Note that the *SSLRequireSSL* directive does not issue a redirect. It merely forbids non-SSL requests.

Or for any version of Apache you can accomplish this using *mod_rewrite*:

```
RewriteEngine On
RewriteCond %{HTTPS} !=on
RewriteRule ^/(.*) https://%{SERVER_NAME}/$1 [R,L]
```

Discussion

It is perhaps best to think of your site's normal pages and its SSL-protected pages as being handled by two separate virtual hosts rather than one. Although they may point to the same content, they run on different ports, are configured differently, and, most important, the browser considers them to be completely separate servers. So you should, too.

Don't think of enabling SSL for a particular directory; rather, you should think of it as redirecting requests for one server to another.

Note that the *Redirect* directive preserves path information, which means that if a request is made for */secure/something.html*, then the redirect will be to *https://secure.example.com/secure/something.html*.

Be careful where you put this directive. Make sure that you only put it in the HTTP (non-SSL) virtual host declaration. Putting it in the global section of the *config* file may cause looping, as the new URL will match the *Redirect* requirement and get redirected itself.

Finally, note that if you want the entire site to be available only *via* SSL, you can accomplish this by simply redirecting all URLs, rather than a particular directory:

```
Redirect / https://secure.example.com/
```

Again, be sure to put that inside the non-SSL virtual host declaration.

You will see various solutions proposed for this situation using *RedirectMatch* or various *RewriteRule* directives. There are special cases in which this is necessary, but in most cases, the simple solution offered here works just fine. In particular, you might be compelled to use this solution when you only have access to your *.htaccess* file, and not to the main server configuration file.

So, the entire setup might look something like this:

```
NameVirtualHost *

<VirtualHost *>
    ServerName regular.example.com
    DocumentRoot /www/docs

    Redirect /secure/ https://secure.example.com/secure/
</VirtualHost>

<VirtualHost _default_:443>
    SSLEngine On
    SSLCertificateFile /www/conf/ssl/ssl.crt
    SSLCertificateKeyFile /www/conf/ssl/ssl.key

    ServerName secure.example.com
    DocumentRoot /www/docs
</VirtualHost>
```

The other two solutions are perhaps more straightforward, although they each have a small additional requirement for use.

The second recipe listed, using *SSLRequireSSL*, will work only if you are using Apache 2.0. It is a directive added specifically to address this need. Placing the *SSLRequireSSL* directive in a particular <*Directory*> section will ensure that non-SSL accesses to that directory are not permitted. It does not redirect users to the SSL host, it merely forbids non-SSL access.

The third recipe, using *RewriteCond* and *RewriteRule* directives, requires that you have *mod_rewrite* installed and enabled. Using the *RewriteCond* directive to check if the client is already using SSL, the *RewriteRule* is invoked only if they are not; in which case, the request is redirected to a request for the same content but using HTTPS instead of HTTP.

See Also

- *http://httpd.apache.org/docs-2.0/mod/mod_ssl.html*
- *http://httpd.apache.org/docs/mod/mod_alias.html*
- *http://httpd.apache.org/docs/mod/mod_rewrite.html*

7.6 Authenticating with Client Certificates

Problem

You want to use client certificates to authenticate access to your site.

Solution

Add the following *mod_ssl* directives to your *httpd.conf* file:

```
SSLVerifyClient require
SSLVerifyDepth 1
SSLCACertificateFile conf/ssl.crt/ca.crt
```

Discussion

If you happen to be lucky enough to have a small, closed user community, such as an intranet, or a Web site for a group of friends or family, it is possible to distribute client certificates so that each user can identify himself.

Create client certificates, signing them with your CA certificate file, and then specify the location of this CA certificate file using the *SSLCACertificateFile* directive, as shown above.

Client certificates are created in the same manner as server certificates, except that the CN (Common Name) on the certificate is the name of the client certificate owner.

See Also

- Recipe 7.3
- *http://httpd.apache.org/docs-2.0/mod/mod_ssl.html*

7.7 SSL Virtual Hosts

Problem

You want to run several SSL hosts on a single IP address.

Solution

There are several possible answers to this problem, depending on your perspective.

The officially correct answer is that you can only run one SSL host per IP address and port. This has to do with the way that SSL works, and is not a limitation specifically of Apache. Attempting to run multiple SSL hosts on the same IP address and port will result in warning messages being displayed by the browser, because it will be receiving the wrong certificate.

One other possible answer is to use a wildcard certificate. This is covered in the next recipe.

Finally, if you don't care about the warning messages, you can set up name-based virtual hosts in the usual way, and simply have Apache use the same certificate for all of them.

In the near future, there will be other solutions to this problem, and there are a considerable number of very smart people working on this problem. Unfortunately, it's still not quite solved.

Discussion

When an https (SSL) request is made by a browser, the first thing that happens is that the certificate is sent to the browser in order for the SSL encryption to be set up. This happens before the browser has told the server what URL it is requesting, so it is impossible to select a particular virtual host. Thus, it is not possible to associate more than one certificate with a particular IP address and port.

There are basically three types of solutions. Either you ignore the problem, you find a way to use one certificate on multiple hostnames, or you use more IP addresses or ports. We'll discuss each of these in turn.

Ignore the problem

In some situations, you may be content to ignore the problem. For test servers, or servers where you have a very small audience, and can explain the situation to each of them, this may be a perfectly acceptable scenario.

In that case, you can set up name-based virtual hosts, and use the same certificate for each one. However, when you connect to any of them, except for the one for which the hostname matches the Common Name on the certificate, you will get a warning message from the browser. In Firefox, this will look like the image below, and will say something like:

You have attempted to establish a connection with `www.example1.com`. However, the security certificate presented belogs to `www.example2.com`. It is possible, although unlikely, that someone may be trying to intercept your communication with this Web site. If you suspect the certificate shown does not belong to `www.example1.com`, please cancel the connection and notify the site administrator.

At this point you, the site administrator, know that everything is fine, and that there's no actual problem. The person on the other end, however, may have any of a number of different reactions. They may panic, and press "Cancel" immediately. They might ignore the message entirely and click "OK," which is almost as bad because ignoring such warning messages is bound to get one into problems eventually. Or they may indeed take the suggested action and contact you, the site administrator. In any of these cases, you probably can immediately see why this isn't a valid solution when you're running an actual secure Web site, doing things like taking credit card transactions.

Use one certificate on several hosts

It is possible to use a single certificate on multiple hostnames, if all of those hostnames are in the same domain. This is called a wildcard certificate, and is discussed in the next recipe.

Use more than one address

The recommended solution is that you run each SSL host on its own IP address. Failing that, you can run different sites on different ports on the same IP address. IP-based virtual hosts are discussed in the chapter on Virtual Hosts, as are port-based virtual hosts. Using either one of these solutions, the warning messages will go away.

If you choose to do port-based virtual hosting, remember that you must put the port number in URLs in order for them to work. For example, if you run your SSL host on port 8443, then you will need to access the site using a URL like:

```
https://www.example.com:8443/
```

This may be inconvenient for the end user, but if access to the site is all through links and form elements, this is a very fine solution.

7.8 Wildcard Certificates

Problem

You want to use a single certificate for multiple hostnames in the same domain.

Solution

Use a wildcard certificate, which works for any name within a particular domain, such as "*.example.com."

Discussion

Using the technique described in Recipe 7.3, create a certificate with a Common Name of `*.example.com`, where `example.com` is the domain for which you wish to use the certificate. This certificate will now work for any hostname in the `example.com` domain, such as `www.example.com` or `secure.example.com`.

On many browsers, however, the certificate will not work for `example.com` or for `one.two.example.com`, but only for hostnames strictly of the form `hostname.example.com`.

Most certificate authorities will charge considerably more to sign wildcard certificates. This is not because it is somehow more complicated to sign these certificates, but is a simple business decision, based on the fact that buying a wildcard certificate means that you don't need to buy multiple single-host certificates.