F.A.Q. List

''The wise man doesn't give the right answers, he poses the right questions.''
Claude Levi-Strauss

Table Of Contents

This chapter is a collection of frequently asked questions (FAQ) and corresponding answers following the popular USENET tradition. Most of these questions occured on the Newsgroup `comp.infosystems.www.servers.unix` or the mod_ssl Support Mailing List `modssl-users@modssl.org`. They are collected at this place to avoid answering the same questions over and over.

Please read this chapter at least once when installing mod_ssl or at least search for your problem here before submitting a problem report to the author.

About the module

- **What is the history of mod_ssl?** [**L**]

  The mod_ssl v1 package was initially created in April 1998 by Ralf S. Engelschall via porting Ben Laurie's Apache-SSL 1.17 source patches for Apache 1.2.6 to Apache 1.3b6. Because of conflicts with Ben Laurie's development cycle it then was re-assembled from scratch for Apache 1.3.0 by merging the old mod_ssl 1.x with the newer Apache-SSL 1.18. From this point on mod_ssl lived its own life as mod_ssl v2. The first publically released version was mod_ssl 2.0.0 from August 10th, 1998. As of this writing (August 1999) the current mod_ssl version is 2.4.0.

  After one year of very active development with over 1000 working hours and over 40 releases mod_ssl reached its current state. The result is an already very clean source base implementing a very rich functionality. The code size increased by a factor of 4 to currently a total of over 10.000 lines of ANSI C consisting of approx. 70% code and 30% code documentation. From the original Apache-SSL code currently approx. 5% is remaining only.

- **What are the functional differences between mod_ssl and Apache-SSL, from where it is originally derived?** [**L**]

  This neither can be answered in short (there were too many code changes) nor can be answered at all by the author (there would immediately be flame wars with no reasonable results at the end). But as you easily can guess from the 5% of remaining Apache-SSL code, a lot of differences exists, although user-visible backward compatibility exists for most things.

  When you really want a detailed comparison you have to read the entries in the large `CHANGES` file that is in the mod_ssl distribution. Usually this is much too hard-core. So I recommend you to either believe in the opinion and recommendations of other users (the simplest approach) or do a comparison yourself (the most reasonable approach). For the latter, grab distributions of mod_ssl (from http://www.modssl.org) and Apache-SSL (from http://www.apache-ssl.org), install both packages, read their documentation and try them out yourself. Then choose the one which pleases you most.

A few final hints to help direct your comparison: quality of documentation ("can you easily find answers and are they sufficient?"), quality of source code ("is the source code reviewable so you can make sure there aren't any trapdoors or inherent security risks because of bad programming style?"), easy and clean installation ("can the SSL functionality easily added to an Apache source tree without manual editing or patching?"), clean integration into Apache ("is the SSL functionality encapsulated and cleanly separated from the remaining Apache functionality?"), support for Dynamic Shared Object (DSO) facility ("can the SSL functionality built as a separate DSO for maximum flexibility?"), Win32 port ("is the SSL functionality available also under the Win32 platform?"), amount and quality of functionality ("is the provided SSL functionality and control possibilities sufficient for your situation?"), quality of problem tracing ("is it possible for you to easily trace down the problems via logfiles, etc?"), etc. pp.

- **What are the major differences between mod_ssl and the commercial alternatives like Raven or Stronghold?** [L]

  In the past (until September 20th, 2000) the major difference was the RSA license which one received (very cheaply in contrast to a direct licensing from RSA DSI) with the commercial Apache SSL products. On the other hand, one needed this license only in the US, of course. So for non-US citizens this point was useless. But now even for US citizens the situations changed because the RSA patent expired on September 20th, 2000 and RSA DSI also placed the RSA algorithm explicitly into the public domain.

  Second, there is the point that one has guaranteed support from the commercial vendors. On the other hand, if you monitored the Open Source quality of mod_ssl and the support activities found on `modssl-users@modssl.org`, you could ask yourself whether you are really convinced that you can get better support from a commercial vendor.

  Third, people often think they would receive perhaps at least a better technical SSL solution than mod_ssl from the commercial vendors. But this is not really true, because all commercial alternatives (Raven 1.4.x, Stronghold 3.x, RedHat SWS 2.x, etc.) *are* actually based on mod_ssl and OpenSSL. The reason for this common misunderstanding is mainly because some vendors make no attempt to make it reasonably clear that their product is actually mod_ssl based. So, do not think, just because the commercial alternatives are usually more expensive, that you are also receiving an alternative *technical* SSL solution. This is usually not the case. Actually the vendor versions of Apache, mod_ssl and OpenSSL often stay behind the latest free versions and perhaps this way still do not include important bug and security fixes. On the other hand, it sometimes occurs that a vendor version includes useful changes which are not available through the official freely available packages. But most vendors play fair and contribute back those changes to the free software world, of course.

So, in short: There are lots of commercial versions of the popular Apache+mod_ssl+OpenSSL server combination available. Every user should decide carefully whether they really need to buy a commercial version or whether it would not be sufficient to directly use the free and official versions of the Apache, mod_ssl and OpenSSL packages.

- **How do I know which mod_ssl version is for which Apache version?** [**L**]

  That's trivial: mod_ssl uses version strings of the syntax *<mod_ssl-version>*-*<apache-version>*, for instance `2.4.0-1.3.9`. This directly indicates that it's mod_ssl version 2.4.0 for Apache version 1.3.9. And this also means you *only* can apply this mod_ssl version to exactly this Apache version (unless you use the `--force` option to mod_ssl's `configure` command ;-).

- **Is mod_ssl Year 2000 compliant?** [**L**]

  Yes, mod_ssl is Year 2000 compliant.

  Because first mod_ssl internally never stores years as two digits. Instead it always uses the ANSI C & POSIX numerical data type `time_t` type, which on almost all Unix platforms at the moment is a `signed long` (usually 32-bits) representing seconds since epoch of January 1st, 1970, 00:00 UTC. This signed value overflows in early January 2038 and not in the year 2000. Second, date and time presentations (for instance the variable ``%{TIME_YEAR}'') are done with full year value instead of abbreviating to two digits.

  Additionally according to a Year 2000 statement from the Apache Group, the Apache webserver is Year 2000 compliant, too. But whether OpenSSL or the underlaying Operating System (either a Unix or Win32 platform) is Year 2000 compliant is a different question which cannot be answered here.

- **What about mod_ssl and the Wassenaar Arrangement?** [**L**]

  First, let us explain what *Wassenaar* and it's *Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies* is: This is a international regime, established 1995, to control trade in conventional arms and dual-use goods and technology. It replaced the previous *CoCom* regime. 33 countries are signatories: Argentina, Australia, Austria, Belgium, Bulgaria, Canada, Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Ireland, Italy, Japan, Luxembourg, Netherlands, New Zealand, Norway, Poland, Portugal, Republic of Korea, Romania, Russian Federation, Slovak Republic, Spain, Sweden, Switzerland, Turkey, Ukraine, United Kingdom and United States. For more details look at http://www.wassenaar.org/.

  In short: The aim of the Wassenaar Arrangement is to prevent the build up of military capabilities that threaten regional and international security and stability. The Wassenaar Arrangement controls the export of cryptography as a dual-use good, i.e., one that has both military and civilian applications. However, the Wassenaar Arrangement also provides an exemption from export controls for mass-market software and free software.

  In the current Wassenaar ``*List of Dual Use Goods and Technologies And Munitions*'', under

‘‘*GENERAL SOFTWARE NOTE*’’ (GSN) it says ‘‘*The Lists do not control "software" which is either: 1. [...] 2. "in the public domain".*’’ And under ‘‘*DEFINITIONS OF TERMS USED IN THESE LISTS*’’ one can find the definition: ‘‘*"In the public domain": This means "technology" or "software" which has been made available without restrictions upon its further dissemination. N.B. Copyright restrictions do not remove "technology" or "software" from being "in the public domain".*’’

So, both mod_ssl and OpenSSL are ‘‘in the public domain’’ for the purposes of the Wassenaar Agreement and its ‘‘*List of Dual Use Goods and Technologies And Munitions List*’’.

Additionally the Wassenaar Agreement itself has no direct consequence for exporting cryptography software. What is actually allowed or forbidden to be exported from the countries has still to be defined in the local laws of each country. And at least according to official press releases from the German BMWi (see here) and the Switzerland Bawi (see here) there will be no forthcoming export restriction for free cryptography software for their countries. Remember that mod_ssl is created in Germany and distributed from Switzerland.

So, mod_ssl and OpenSSL are not affected by the Wassenaar Agreement.

About Installation

- **When I access my website the first time via HTTPS I get a core dump?** [L]

  There can be a lot of reasons why a core dump can occur, of course. Ranging from buggy third-party modules, over buggy vendor libraries up to a buggy mod_ssl version. But the above situation is often caused by old or broken vendor DBM libraries. To solve it either build mod_ssl with the built-in SDBM library (specify `--enable-rule=SSL_SDBM` at the APACI command line) or switch from ‘‘`SSLSessionCache dbm:`’’ to the newer ‘‘`SSLSessionCache shm:`’’ variant (after you have rebuilt Apache with MM, of course).

- **My Apache dumps core when I add both mod_ssl and PHP3?** [L]

  Make sure you add mod_ssl to the Apache source tree first and then do a fresh configuration and installation of PHP3. For SSL support EAPI patches are required which have to change internal Apache structures. PHP3 needs to know about these in order to work correctly. Always make sure that `-DEAPI` is contained in the compiler flags when PHP3 is build.

- **When I startup Apache I get errors about undefined symbols like ap_global_ctx?** [L]

  This actually means you installed mod_ssl as a DSO, but without rebuilding Apache with EAPI. Because EAPI is a requirement for mod_ssl, you need an extra patched Apache (containing the EAPI patches) and you have to build this Apache with EAPI enabled (explicitly specify `--enable-rule=EAPI` at the APACI command line).

- **When I startup Apache I get permission errors related to SSLMutex?** [L]

  When you receive entries like ‘‘`mod_ssl: Child could not open SSLMutex lockfile`

`/opt/apache/logs/ssl_mutex.18332 (System error follows) [...] System: Permission denied (errno: 13)`'' this is usually caused by to restrictive permissions on the *parent* directories. Make sure that all parent directories (here `/opt`, `/opt/apache` and `/opt/apache/logs`) have the x-bit set at least for the UID under which Apache's children are running (see the `User` directive of Apache).

- **When I use the MM library and the shared memory cache each process grows 1.5MB according to 'top' although I specified 512000 as the cache size?** [L]

  The additional 1MB are caused by the global shared memory pool EAPI allocates for all modules and which is not used by mod_ssl for various reasons. So the actually allocated shared memory is always 1MB more than what you specify on `SSLSessionCache`. But don't be confused by the display of 'top': although is indicates that *each* process grow, this is not reality, of course. Instead the additional memory consumption is shared by all processes, i.e. the 1.5MB are allocated only once per Apache instance and not once per Apache server process.

- **Apache creates files in a directory declared by the internal EAPI_MM_CORE_PATH define. Is there a way to override the path using a configuration directive?** [L]

  No, there is not configuration directive, because for technical bootstrapping reasons, a directive not possible at all. Instead use ``CFLAGS='-DEAPI_MM_CORE_PATH="/path/to/wherever/"' ./configure ...`' when building Apache or use option **-d** when starting `httpd`.

- **When I fire up the server, mod_ssl stops with the error "Failed to generate temporary 512 bit RSA private key", why? And a "PRNG not seeded" error occurs if I try "make certificate".** [L]

  Cryptographic software needs a source of unpredictable data to work correctly. Many open source operating systems provide a "randomness device" that serves this purpose (usually named `/dev/random`). On other systems, applications have to seed the OpenSSL Pseudo Random Number Generator (PRNG) manually with appropriate data before generating keys or performing public key encryption. As of version 0.9.5, the OpenSSL functions that need randomness report an error if the PRNG has not been seeded with at least 128 bits of randomness. So mod_ssl has to provide enough entropy to the PRNG to work correctly. For this one has to use the `SSLRandomSeed` directives (to solve the run-time problem) and create a `$HOME/.rnd` file to make sure enough entropy is available also for the "`make certificate`" step (in case the "`make certificate`" procedure is not able to gather enough entropy theirself by searching for system files).

About Configuration

- **Is it possible to provide HTTP and HTTPS with a single server?** [L]

  Yes, HTTP and HTTPS use different server ports, so there is no direct conflict between them. Either run two separate server instances (one binds to port 80, the other to port 443) or even use Apache's elegant virtual hosting facility where you can easily create two virtual servers which Apache dispatches: one responding to port 80 and speaking HTTP and one responding to port 443 speaking HTTPS.

- **I know that HTTP is on port 80, but where is HTTPS?** [**L**]

  You can run HTTPS on any port, but the standards specify port 443, which is where any HTTPS compliant browser will look by default. You can force your browser to look on a different port by specifying it in the URL like this (for port 666):
  `https://secure.server.dom:666/`

- **How can I speak HTTPS manually for testing purposes?** [**L**]

  While you usually just use

  ```
  $ telnet localhost 80
  GET / HTTP/1.0
  ```

  for simple testing the HTTP protocol of Apache, it's not such easy for HTTPS because of the SSL protocol between TCP and HTTP. But with the help of OpenSSL's `s_client` command you can do a similar check even for HTTPS:

  ```
  $ openssl s_client -connect localhost:443 -state -debug
  GET / HTTP/1.0
  ```

  Before the actual HTTP response you receive detailed information about the SSL handshake. For a more general command line client which directly understands both the HTTP and HTTPS scheme, can perform GET and POST methods, can use a proxy, supports byte ranges, etc. you should have a look at nifty cURL tool. With it you can directly check if your Apache is running fine on Port 80 and 443 as following:

  ```
  $ curl http://localhost/
  $ curl https://localhost/
  ```

- **Why does the connection hang when I connect to my SSL-aware Apache server?** [**L**]

  Because you connected with HTTP to the HTTPS port, i.e. you used an URL of the form ''`http://`'' instead of ''`https://`''. This also happens the other way round when you connect via HTTPS to a HTTP port, i.e. when you try to use ''`https://`'' on a server that doesn't support SSL (on this port). Make sure you are connecting to a virtual server that supports SSL, which is probably the IP associated with your hostname, not localhost (127.0.0.1).

- **Why do I get ''Connection Refused'' messages when trying to access my freshly installed Apache+mod_ssl server via HTTPS?** [**L**]

  There can be various reasons. Some of the common mistakes is that people start Apache with just ''`apachectl start`'' (or ''`httpd`'') instead of ''`apachectl startssl`'' (or ''`httpd -DSSL`''. Or you're configuration is not correct. At least make sure that your ''`Listen`'' directives match your ‘‘<VirtualHost>’’ directives. And if all fails, please do yourself a favor and start over with the default configuration mod_ssl provides you.

- **In my CGI programs and SSI scripts the various documented SSL_XXX variables do not exists. Why?** [**L**]

  Just make sure you have ‘‘SSLOptions +StdEnvVars’’ enabled for the context of your CGI/SSI requests.

- **How can I use relative hyperlinks to switch between HTTP and HTTPS?** [**L**]

  Usually you have to use fully-qualified hyperlinks because you have to change the URL scheme. But with the help of some URL manipulations through mod_rewrite you can achieve the same effect while you still can use relative URLs:

```
RewriteEngine on
RewriteRule   ^/(.*):SSL$   https://%{SERVER_NAME}/$1 [R,L]
RewriteRule   ^/(.*):NOSSL$ http://%{SERVER_NAME}/$1  [R,L]
```

  This rewrite ruleset lets you use hyperlinks of the form

```
<a href="document.html:SSL">
```

About Certificates

- **What are RSA Private Keys, CSRs and Certificates?** [**L**]

  The RSA private key file is a digital file that you can use to decrypt messages sent to you. It has a public component which you distribute (via your Certificate file) which allows people to encrypt those messages to you. A Certificate Signing Request (CSR) is a digital file which contains your public key and your name. You send the CSR to a Certifying Authority (CA) to be converted into a real Certificate. A Certificate contains your RSA public key, your name, the name of the CA, and is digitally signed by your CA. Browsers that know the CA can verify the signature on that Certificate, thereby obtaining your RSA public key. That enables them to send messages which only you can decrypt. See the Introduction chapter for a general description of the SSL protocol.

- **Seems like there is a difference on startup between the original Apache and an SSL-aware Apache?** [**L**]

  Yes, in general, starting Apache with a built-in mod_ssl is just like starting an unencumbered Apache, except for the fact that when you have a pass phrase on your SSL private key file. Then a startup dialog pops up asking you to enter the pass phrase.

  To type in the pass phrase manually when starting the server can be problematic, for instance when starting the server from the system boot scripts. As an alternative to this situation you can follow the steps below under ‘‘How can I get rid of the pass-phrase dialog at Apache startup time?’’.

- **How can I create a dummy SSL server Certificate for testing purposes?** [**L**]

  A Certificate does not have to be signed by a public CA. You can use your private key to sign the Certificate which contains your public key. You can install this Certificate into your server, and people using Netscape Navigator (not MSIE) will be able to connect after clicking OK to a warning dialogue. You can get MSIE to work, and your customers can eliminate the dialogue, by installing that Certificate manually into their browsers.

  Just use the ‘‘`make certificate`’’ command at the top-level directory of the Apache source tree right before installing Apache via ‘‘`make install`’’. This creates a self-signed SSL Certificate which expires after 30 days and isn’t encrypted (which means you don’t need to enter a pass-phrase at Apache startup time).

  BUT REMEMBER: YOU REALLY HAVE TO CREATE A REAL CERTIFICATE FOR THE LONG RUN! HOW THIS IS DONE IS DESCRIBED IN THE NEXT ANSWER.

- **Ok, I’ve got my server installed and want to create a real SSL server Certificate for it. How do I do it?** [**L**]

  Here is a step-by-step description:

  1. Make sure OpenSSL is really installed and in your `PATH`. But some commands even work ok when you just run the ‘‘`openssl`’’ program from within the OpenSSL source tree as ‘‘`./apps/openssl`’’.

  2. Create a RSA private key for your Apache server (will be Triple-DES encrypted and PEM formatted):

     ```
     $ openssl genrsa -des3 -out server.key 1024
     ```

     Please backup this `server.key` file and remember the pass-phrase you had to enter at a secure location. You can see the details of this RSA private key via the command:

     ```
     $ openssl rsa -noout -text -in server.key
     ```

     And you could create a decrypted PEM version (not recommended) of this RSA private key via:

     ```
     $ openssl rsa -in server.key -out server.key.unsecure
     ```

  3. Create a Certificate Signing Request (CSR) with the server RSA private key (output will be PEM formatted):

     ```
     $ openssl req -new -key server.key -out server.csr
     ```

Make sure you enter the FQDN ("Fully Qualified Domain Name") of the server when OpenSSL prompts you for the "CommonName", i.e. when you generate a CSR for a website which will be later accessed via `https://www.foo.dom/`, enter "www.foo.dom" here. You can see the details of this CSR via the command

```
$ openssl req -noout -text -in server.csr
```

4. You now have to send this Certificate Signing Request (CSR) to a Certifying Authority (CA) for signing. The result is then a real Certificate which can be used for Apache. Here you have to options: First you can let the CSR sign by a commercial CA like Verisign or Thawte. Then you usually have to post the CSR into a web form, pay for the signing and await the signed Certificate you then can store into a server.crt file. For more information about commercial CAs have a look at the following locations:

    - Verisign
      http://digitalid.verisign.com/server/apacheNotice.htm
    - Thawte Consulting
      http://www.thawte.com/certs/server/request.html
    - CertiSign Certificadora Digital Ltda.
      http://www.certisign.com.br
    - IKS GmbH
      http://www.iks-jena.de/produkte/ca/
    - Uptime Commerce Ltd.
      http://www.uptimecommerce.com
    - BelSign NV/SA
      http://www.belsign.be

    Second you can use your own CA and now have to sign the CSR yourself by this CA. Read the next answer in this FAQ on how to sign a CSR with your CA yourself. You can see the details of the received Certificate via the command:

    ```
    $ openssl x509 -noout -text -in server.crt
    ```

5. Now you have two files: `server.key` and `server.crt`. These now can be used as following inside your Apache's `httpd.conf` file:

    ```
           SSLCertificateFile    /path/to/this/server.crt
           SSLCertificateKeyFile /path/to/this/server.key
    ```

    The `server.csr` file is no longer needed.

- **How can I create and use my own Certificate Authority (CA)?** [L]

  The short answer is to use the `CA.sh` or `CA.pl` script provided by OpenSSL. The long and manual answer is this:

  1. Create a RSA private key for your CA (will be Triple-DES encrypted and PEM formatted):

```
$ openssl genrsa -des3 -out ca.key 1024
```

Please backup this `ca.key` file and remember the pass-phrase you currently entered at a secure location. You can see the details of this RSA private key via the command

```
$ openssl rsa -noout -text -in ca.key
```

And you can create a decrypted PEM version (not recommended) of this private key via:

```
$ openssl rsa -in ca.key -out ca.key.unsecure
```

2. Create a self-signed CA Certificate (X509 structure) with the RSA key of the CA (output will be PEM formatted):

```
$ openssl req -new -x509 -days 365 -key ca.key -out ca.crt
```

You can see the details of this Certificate via the command:

```
$ openssl x509 -noout -text -in ca.crt
```

3. Prepare a script for signing which is needed because the ‘‘`openssl ca`’’ command has some strange requirements and the default OpenSSL config doesn’t allow one easily to use ‘‘`openssl ca`’’ directly. So a script named `sign.sh` is distributed with the mod_ssl distribution (subdir `pkg.contrib/`). Use this script for signing.

4. Now you can use this CA to sign server CSR’s in order to create real SSL Certificates for use inside an Apache webserver (assuming you already have a `server.csr` at hand):

```
$ ./sign.sh server.csr
```

This signs the server CSR and results in a `server.crt` file.

- **How can I change the pass-phrase on my private key file?** [L]

You simply have to read it with the old pass-phrase and write it again by specifying the new pass-phrase. You can accomplish this with the following commands:

```
$ openssl rsa -des3 -in server.key -out server.key.new
$ mv server.key.new server.key
```

Here you’re asked two times for a PEM pass-phrase. At the first prompt enter the old pass-phrase and at the second prompt enter the new pass-phrase.

- **How can I get rid of the pass-phrase dialog at Apache startup time?** [L]

The reason why this dialog pops up at startup and every re-start is that the RSA private key inside your server.key file is stored in encrypted format for security reasons. The pass-phrase is needed to be able to read and parse this file. When you can be sure that your server is secure enough you perform two steps:

1. Remove the encryption from the RSA private key (while preserving the original file):

   ```
   $ cp server.key server.key.org
   $ openssl rsa -in server.key.org -out server.key
   ```

2. Make sure the server.key file is now only readable by root:

   ```
   $ chmod 400 server.key
   ```

Now `server.key` will contain an unencrypted copy of the key. If you point your server at this file it will not prompt you for a pass-phrase. HOWEVER, if anyone gets this key they will be able to impersonate you on the net. PLEASE make sure that the permissions on that file are really such that only root or the web server user can read it (preferably get your web server to start as root but run as another server, and have the key readable only by root).

As an alternative approach you can use the ‘‘`SSLPassPhraseDialog exec:/path/to/program`’’ facility. But keep in mind that this is neither more nor less secure, of course.

- **How do I verify that a private key matches its Certificate?** [**L**]

The private key contains a series of numbers. Two of those numbers form the "public key", the others are part of your "private key". The "public key" bits are also embedded in your Certificate (we get them from your CSR). To check that the public key in your cert matches the public portion of your private key, you need to view the cert and the key and compare the numbers. To view the Certificate and the key run the commands:

```
$ openssl x509 -noout -text -in server.crt
$ openssl rsa -noout -text -in server.key
```

The ‘modulus’ and the ‘public exponent’ portions in the key and the Certificate must match. But since the public exponent is usually 65537 and it’s bothering comparing long modulus you can use the following approach:

```
$ openssl x509 -noout -modulus -in server.crt | openssl md5
$ openssl rsa -noout -modulus -in server.key | openssl md5
```

And then compare these really shorter numbers. With overwhelming probability they will differ if the keys are different. BTW, if I want to check to which key or certificate a particular CSR belongs you can compute

```
$ openssl req -noout -modulus -in server.csr | openssl md5
```

- **What does it mean when my connections fail with an "alert bad certificate" error?** [**L**]

Usually when you see errors like ‘‘`OpenSSL: error:14094412: SSL routines:SSL3_READ_BYTES:sslv3 alert bad certificate`’’ in the SSL logfile, this means that the browser was unable to handle the server certificate/private-key which perhaps contain a RSA-key not equal to 1024 bits. For instance Netscape Navigator 3.x is one of those browsers.

- **Why does my 2048-bit private key not work?** [**L**]

  The private key sizes for SSL must be either 512 or 1024 for compatibility with certain web browsers. A keysize of 1024 bits is recommended because keys larger than 1024 bits are incompatible with some versions of Netscape Navigator and Microsoft Internet Explorer, and with other browsers that use RSA's BSAFE cryptography toolkit.

- **Why is client authentication broken after upgrading from SSLeay version 0.8 to 0.9?** [**L**]

  The CA certificates under the path you configured with `SSLCACertificatePath` are found by SSLeay through hash symlinks. These hash values are generated by the '`openssl x509 -noout -hash`' command. But the algorithm used to calculate the hash for a certificate has changed between SSLeay 0.8 and 0.9. So you have to remove all old hash symlinks and re-create new ones after upgrading. Use the `Makefile` mod_ssl placed into this directory.

- **How can I convert a certificate from PEM to DER format?** [**L**]

  The default certificate format for SSLeay/OpenSSL is PEM, which actually is Base64 encoded DER with header and footer lines. For some applications (e.g. Microsoft Internet Explorer) you need the certificate in plain DER format. You can convert a PEM file `cert.pem` into the corresponding DER file `cert.der` with the following command: **`$ openssl x509 -in cert.pem -out cert.der -outform DER`**

- **I try to install a Verisign certificate. Why can't I find neither the `getca` nor `getverisign` programs Verisign mentions?** [**L**]

  This is because Verisign has never provided specific instructions for Apache+mod_ssl. Rather they tell you what you should do if you were using C2Net's Stronghold (a commercial Apache based server with SSL support). The only thing you have to do is to save the certificate into a file and give the name of that file to the `SSLCertificateFile` directive. Remember that you need to give the key file in as well (see `SSLCertificateKeyFile` directive). For a better CA-related overview on SSL certificate fiddling you can look at Thawte's mod_ssl instructions.

- **Can I use the Server Gated Cryptography (SGC) facility (aka Verisign Global ID) also with mod_ssl?** [**L**]

  Yes, mod_ssl since version 2.1 supports the SGC facility. You don't have to configure anything special for this, just use a Global ID as your server certificate. The *step up* of the clients are then automatically handled by mod_ssl under run-time. For details please read the `README.GlobalID` document in the mod_ssl distribution.

- **After I have installed my new Verisign Global ID server certificate, the browsers**

complain that they cannot verify the server certificate? [**L**]

That is because Verisign uses an intermediate CA certificate between the root CA certificate (which is installed in the browsers) and the server certificate (which you installed in the server). You should have received this additional CA certificate from Verisign. If not, complain to them. Then configure this certificate with the `SSLCertificateChainFile` directive in the server. This makes sure the intermediate CA certificate is send to the browser and this way fills the gap in the certificate chain.

About SSL Protocol

- **Why has my webserver a higher load now that I run SSL there?** [**L**]

  Because SSL uses strong cryptographic encryption and this needs a lot of number crunching. And because when you request a webpage via HTTPS even the images are transfered encrypted. So, when you have a lot of HTTPS traffic the load increases.

- **Often HTTPS connections to my server require up to 30 seconds for establishing the connection, although sometimes it works faster?** [**L**]

  Usually this is caused by using a `/dev/random` device for `SSLRandomSeed` which is blocking in read(2) calls if not enough entropy is available. Read more about this problem in the refernce chapter under `SSLRandomSeed`.

- **What SSL Ciphers are supported by mod_ssl?** [**L**]

  Usually just all SSL ciphers which are supported by the version of OpenSSL in use (can depend on the way you built OpenSSL). Typically this at least includes the following:

  - RC4 with MD5
  - RC4 with MD5 (export version restricted to 40-bit key)
  - RC2 with MD5
  - RC2 with MD5 (export version restricted to 40-bit key)
  - IDEA with MD5
  - DES with MD5
  - Triple-DES with MD5

  To determine the actual list of supported ciphers you can run the following command:

  ```
  $ openssl ciphers -v
  ```

- **I want to use Anonymous Diffie-Hellman (ADH) ciphers, but I always get ‘‘no shared cipher’’ errors?** [**L**]

  In order to use Anonymous Diffie-Hellman (ADH) ciphers, it is not enough to just put ‘‘`ADH`’’ into your `SSLCipherSuite`. Additionally you have to build OpenSSL with ‘‘`-DSSL_ALLOW_ADH`’’. Because per default OpenSSL does not allow ADH ciphers for security reasons. So if you are actually enabling these ciphers make sure you are informed about the side-effects.

- **I always just get a ’no shared ciphers’ error if I try to connect to my freshly installed server?** [**L**]

  Either you have messed up your `SSLCipherSuite` directive (compare it with the pre-configured example in `httpd.conf-dist`) or you have choosen the DSA/DH algorithms instead of RSA under "`make certificate`" and ignored or overseen the warnings. Because if you have choosen DSA/DH, then your server no longer speaks RSA-based SSL ciphers (at least not until you also configure an additional RSA-based certificate/key pair). But current browsers like NS or IE only speak RSA ciphers. The result is the "no shared ciphers" error. To fix this, regenerate your server certificate/key pair and this time choose the RSA algorithm.

- **Why can’t I use SSL with name-based/non-IP-based virtual hosts?** [**L**]

  The reason is very technical. Actually it’s some sort of a chicken and egg problem: The SSL protocol layer stays below the HTTP protocol layer and encapsulates HTTP. When an SSL connection (HTTPS) is established Apache/mod_ssl has to negotiate the SSL protocol parameters with the client. For this mod_ssl has to consult the configuration of the virtual server (for instance it has to look for the cipher suite, the server certificate, etc.). But in order to dispatch to the correct virtual server Apache has to know the `Host` HTTP header field. For this the HTTP request header has to be read. This cannot be done before the SSL handshake is finished. But the information is already needed at the SSL handshake phase. Bingo!

- **When I use Basic Authentication over HTTPS the lock icon in Netscape browsers still show the unlocked state when the dialog pops up. Does this mean the username/password is still transmitted unencrypted?** [**L**]

  No, the username/password is already transmitted encrypted. The icon in Netscape browsers is just not really synchronized with the SSL/TLS layer (it toggles to the locked state when the first part of the actual webpage data is transferred which is not quite correct) and this way confuses people. The Basic Authentication facility is part of the HTTP layer and this layer is above the SSL/TLS layer in HTTPS. And before any HTTP data communication takes place in HTTPS the SSL/TLS layer has already done the handshake phase and switched to encrypted communication. So, don’t get confused by this icon.

- **When I connect via HTTPS to an Apache+mod_ssl+OpenSSL server with Microsoft Internet Explorer (MSIE) I get various I/O errors. What is the reason?** [**L**]

  The first reason is that the SSL implementation in some MSIE versions has some subtle bugs related to the HTTP keep-alive facility and the SSL close notify alerts on socket connection close. Additionally the interaction between SSL and HTTP/1.1 features are problematic with some MSIE versions, too. You’ve to work-around these problems by forcing Apache+mod_ssl+OpenSSL to not use HTTP/1.1, keep-alive connections or sending the SSL close notify messages to MSIE clients. This can be done by using the following directive in your SSL-aware virtual host section:

```
SetEnvIf User-Agent ".*MSIE.*" \
         nokeepalive ssl-unclean-shutdown \
         downgrade-1.0 force-response-1.0
```

Additionally it is known some MSIE versions have also problems with particular ciphers. Unfortunately one cannot workaround these bugs only for those MSIE particular clients, because the ciphers are already used in the SSL handshake phase. So a MSIE-specific `SetEnvIf` doesn't work to solve these problems. Instead one has to do more drastic adjustments to the global parameters. But before you decide to do this, make sure your clients really have problems. If not, do not do this, because it affects all(!) your clients, i.e., also your non-MSIE clients.

The next problem is that 56bit export versions of MSIE 5.x browsers have a broken SSLv3 implementation which badly interacts with OpenSSL versions greater than 0.9.4. You can either accept this and force your clients to upgrade their browsers, or you downgrade to OpenSSL 0.9.4 (hmmm), or you can decide to workaround it by accepting the drawback that your workaround will horribly affect also other browsers:

```
SSLProtocol all -SSLv3
```

This completely disables the SSLv3 protocol and lets those browsers work. But usually this is an even less acceptable workaround. A more reasonable workaround is to address the problem more closely and disable only the ciphers which cause trouble.

```
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

This also lets the broken MSIE versions work, but only removes the newer 56bit TLS ciphers.

Another problem with MSIE 5.x clients is that they refuse to connect to URLs of the form `https://12.34.56.78/` (IP-addresses are used instead of the hostname), if the server is using the Server Gated Cryptography (SGC) facility. This can only be avoided by using the fully qualified domain name (FQDN) of the website in hyperlinks instead, because MSIE 5.x has an error in the way it handles the SGC negotiation.

- **When I connect via HTTPS to an Apache+mod_ssl server with Netscape Navigator I get I/O errors and the message "Netscape has encountered bad data from the server" What's the reason?** [L]

  The problem usually is that you had created a new server certificate with the same DN, but you had told your browser to accept forever the old server certificate. Once you clear the entry in your browser for the old certificate, everything usually will work fine. Netscape's SSL implementation is correct, so when you encounter I/O errors with Netscape Navigator it is most of the time caused by the configured certificates.

About Support

- **What information resources are available in case of mod_ssl problems?** [L]

The following information resources are available. In case of problems you should search here first.

1. *Answers in the User Manual's F.A.Q. List (this)*
   http://www.modssl.org/docs/2.7/ssl_faq.html
   First look inside the F.A.Q. (this text), perhaps your problem is such popular that it was already answered a lot of times in the past.

2. *Postings from the modssl-users Support Mailing List* http://www.modssl.org/support/
   Second search for your problem in one of the existing archives of the modssl-users mailing list. Perhaps your problem popped up at least once for another user, too.

3. *Problem Reports in the Bug Database* http://www.modssl.org/support/bugdb/
   Third look inside the mod_ssl Bug Database. Perhaps someone else already has reported the problem.

- **What support contacts are available in case of mod_ssl problems?** [**L**]

  The following lists all support possibilities for mod_ssl, in order of preference, i.e. start in this order and do not pick the support possibility you just like most, please.

  1. *Write a Problem Report into the Bug Database*
     http://www.modssl.org/support/bugdb/
     This is the preferred way of submitting your problem report, because this way it gets filed into the bug database (it cannot be lost) *and* send to the modssl-users mailing list (others see the current problems and learn from answers).

  2. *Write a Problem Report to the modssl-users Support Mailing List*
     modssl-users @ modssl.org
     This is the second way of submitting your problem report. You have to subscribe to the list first, but then you can easily discuss your problem with both the author and the whole mod_ssl user community.

  3. *Write a Problem Report to the author*
     rse @ engelschall.com
     This is the last way of submitting your problem report. Please avoid this in your own interest because the author is really a very busy men. Your mail will always be filed to one of his various mail-folders and is usually not processed as fast as a posting on modssl-users.

- **What information and details I've to provide to the author when writing a bug report?** [**L**]

  You have to at least always provide the following information:

  - *Apache, mod_ssl and OpenSSL version information*
    The mod_ssl version you should really know. For instance, it's the version number in the distribution tarball. The Apache version can be determined by running ``` ``httpd -v'' ```.

The OpenSSL version can be determined by running ‘‘`openssl version`’’. Alternatively when you have Lynx installed you can run the command ‘‘`lynx -mime_header http://localhost/ | grep Server`’’ to determine all information in a single step.

- *The details on how you built and installed Apache+mod_ssl+OpenSSL*
  For this you can provide a logfile of your terminal session which shows the configuration and install steps. Alternatively you can at least provide the author with the APACI ‘`configure`’’ command line you used (assuming you used APACI, of course).

- *In case of core dumps please include a Backtrace*
  In case your Apache+mod_ssl+OpenSSL should really dumped core please attach a stack-frame ‘‘backtrace’’ (see the next question on how to get it). Without this information the reason for your core dump cannot be found. So you have to provide the backtrace, please.

- *A detailed description of your problem*
  Don’t laugh, I’m totally serious. I already got a lot of problem reports where the people not really said what’s the actual problem is. So, in your own interest (you want the problem be solved, don’t you?) include as much details as possible, please. But start with the essentials first, of course.

- **I got a core dump, can you help me?** [L]

  In general no, at least not unless you provide more details about the code location where Apache dumped core. What is usually always required in order to help you is a backtrace (see next question). Without this information it is mostly impossible to find the problem and help you in fixing it.

- **Ok, I got a core dump but how do I get a backtrace to find out the reason for it?** [L]

  Follow the following steps:

  1. Make sure you have debugging symbols available in at least Apache and mod_ssl. On platforms where you use GCC/GDB you have to build Apache+mod_ssl with ‘‘`OPTIM="-g -ggdb3"`’’ to achieve this. On other platforms at least ‘‘`OPTIM="-g"`’’ is needed.

  2. Startup the server and try to produce the core-dump. For this you perhaps want to use a directive like ‘‘`CoreDumpDirectory /tmp`’’ to make sure that the core-dump file can be written. You then should get a `/tmp/core` or `/tmp/httpd.core` file. When you don’t get this, try to run your server under an UID != 0 (root), because most "current" kernels do not allow a process to dump core after it has done a `setuid()` (unless it does an `exec()`) for security reasons (there can be privileged information left over in memory). Additionally you can run ‘‘`/path/to/httpd -X`’’ manually to force Apache to not fork.

  3. Analyze the core-dump. For this run ‘‘`gdb /path/to/httpd /tmp/httpd.core`’’ or a similar command has to run. In GDB you then just have to enter the ‘‘`bt`’’ command

and, voila, you get the backtrace. For other debuggers consult your local debugger manual. Send this backtrace to the author.