Apache Module mod_ssl

Description:	Strong cryptography using the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols
Status:	Extension
Module Identifier:	ssl_module
Source File:	mod_ssl.c

Summary

This module provides SSL v2/v3 and TLS v1 support for the Apache HTTP Server. It was contributed by Ralf S. Engeschall based on his mod_ssl project and originally derived from work by Ben Laurie.

This module relies on OpenSSL[1] to provide the cryptography engine.

Further details, discussion, and examples are provided in the SSL documentation[2].

Topics

Directives

Environment Variables

This module provides a lot of SSL information as additional environment variables to the SSI and CGI namespace. The generated variables are listed in the table below. For backward compatibility the information can be made available under different names, too. Look in the Compatibility[3] chapter for details on the compatibility variables.

Variable Name:	Value Type:	Description:
`HTTPS`	flag	HTTPS is being used.
`SSL_PROTOCOL`	string	The SSL protocol version (SSLv2, SSLv3, TLSv1)
`SSL_SESSION_ID`	string	The hex-encoded SSL session id
`SSL_CIPHER`	string	The cipher specification name
`SSL_CIPHER_EXPORT`	string	`true` if cipher is an export cipher

Variable Name:	**Value Type:**	**Description:**
`SSL_CIPHER_USEKEYSIZE`	number	Number of cipher bits (actually used)
`SSL_CIPHER_ALGKEYSIZE`	number	Number of cipher bits (possible)
`SSL_VERSION_INTERFACE`	string	The mod_ssl program version
`SSL_VERSION_LIBRARY`	string	The OpenSSL program version
`SSL_CLIENT_M_VERSION`	string	The version of the client certificate
`SSL_CLIENT_M_SERIAL`	string	The serial of the client certificate
`SSL_CLIENT_S_DN`	string	Subject DN in client's certificate
`SSL_CLIENT_S_DN_`*x509*	string	Component of client's Subject DN
`SSL_CLIENT_I_DN`	string	Issuer DN of client's certificate
`SSL_CLIENT_I_DN_`*x509*	string	Component of client's Issuer DN
`SSL_CLIENT_V_START`	string	Validity of client's certificate (start time)
`SSL_CLIENT_V_END`	string	Validity of client's certificate (end time)
`SSL_CLIENT_A_SIG`	string	Algorithm used for the signature of client's certificate
`SSL_CLIENT_A_KEY`	string	Algorithm used for the public key of client's certificate
`SSL_CLIENT_CERT`	string	PEM-encoded client certificate
`SSL_CLIENT_CERT_CHAIN`*n*	string	PEM-encoded certificates in client certificate chain
`SSL_CLIENT_VERIFY`	string	`NONE`, `SUCCESS`, `GENEROUS` or `FAILED:`*reason*
`SSL_SERVER_M_VERSION`	string	The version of the server certificate
`SSL_SERVER_M_SERIAL`	string	The serial of the server certificate
`SSL_SERVER_S_DN`	string	Subject DN in server's certificate
`SSL_SERVER_S_DN_`*x509*	string	Component of server's Subject DN
`SSL_SERVER_I_DN`	string	Issuer DN of server's certificate
`SSL_SERVER_I_DN_`*x509*	string	Component of server's Issuer DN
`SSL_SERVER_V_START`	string	Validity of server's certificate (start time)
`SSL_SERVER_V_END`	string	Validity of server's certificate (end time)
`SSL_SERVER_A_SIG`	string	Algorithm used for the signature of server's certificate
`SSL_SERVER_A_KEY`	string	Algorithm used for the public key of server's certificate
`SSL_SERVER_CERT`	string	PEM-encoded server certificate
[where *x509* is a component of a X.509 DN: `C,ST,L,O,OU,CN,T,I,G,S,D,UID,Email`]		

Custom Log Formats

When `mod_ssl` is built into Apache or at least loaded (under DSO situation) additional functions exist for the Custom Log Format [4] of `mod_log_config`. First there is an additional ``%{*varname*}x" eXtension format function which can be used to expand any variables provided by any module, especially those provided by mod_ssl which can you find in the above table.

For backward compatibility there is additionally a special ``%{*name*}c" cryptography format function provided. Information about this function is provided in the Compatibility[3] chapter.

Example:

```
CustomLog logs/ssl_request_log \ "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x
\"%r\" %b"
```

SSLCACertificateFile Directive

Description:	File of concatenated PEM-encoded CA Certificates for Client Auth
Syntax:	`SSLCACertificateFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the *all-in-one* file where you can assemble the Certificates of Certification Authorities (CA) whose *clients* you deal with. These are used for Client Authentication. Such a file is simply the concatenation of the various PEM-encoded Certificate files, in order of preference. This can be used alternatively and/or additionally to `SSLCACertificatePath`.

Example

```
SSLCACertificateFile
/usr/local/apache/conf/ssl.crt/ca-bundle-client.crt
```

SSLCACertificatePath Directive

Description:	Directory of PEM-encoded CA Certificates for Client Auth
Syntax:	`SSLCACertificatePath` *`directory-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the directory where you keep the Certificates of Certification Authorities (CAs) whose clients you deal with. These are used to verify the client certificate on Client Authentication.

The files in this directory have to be PEM-encoded and are accessed through hash filenames. So usually you can't just place the Certificate files there: you also have to create symbolic links named *hash-value*`.N`. And you should always make sure this directory contains the appropriate symbolic links. Use the `Makefile` which comes with mod_ssl to accomplish this task.

Example

```
SSLCACertificatePath /usr/local/apache/conf/ssl.crt/
```

SSLCARevocationFile Directive

Description:	File of concatenated PEM-encoded CA CRLs for Client Auth
Syntax:	`SSLCARevocationFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the *all-in-one* file where you can assemble the Certificate Revocation Lists (CRL) of Certification Authorities (CA) whose *clients* you deal with. These are used for Client Authentication. Such a file is simply the concatenation of the various PEM-encoded CRL files, in order of preference. This can be used alternatively and/or additionally to `SSLCARevocationPath`.

Example
```
SSLCARevocationFile /usr/local/apache/conf/ssl.crl/ca-bundle-client.crl
```

SSLCARevocationPath Directive

Description:	Directory of PEM-encoded CA CRLs for Client Auth
Syntax:	`SSLCARevocationPath` *`directory-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the directory where you keep the Certificate Revocation Lists (CRL) of Certification Authorities (CAs) whose clients you deal with. These are used to revoke the client certificate on Client Authentication.

The files in this directory have to be PEM-encoded and are accessed through hash filenames. So usually you have not only to place the CRL files there. Additionally you have to create symbolic links named *hash-value*`.rN`. And you should always make sure this directory contains the appropriate symbolic links. Use the `Makefile` which comes with `mod_ssl` to accomplish this task.

Example
```
SSLCARevocationPath /usr/local/apache/conf/ssl.crl/
```

SSLCertificateChainFile Directive

Description:	File of PEM-encoded Server CA Certificates
Syntax:	`SSLCertificateChainFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the optional *all-in-one* file where you can assemble the certificates of Certification Authorities (CA) which form the certificate chain of the server certificate. This starts with the issuing CA certificate of of the server certificate and can range up to the root CA certificate. Such a file is simply the concatenation of the various PEM-encoded CA Certificate files, usually in certificate chain order.

This should be used alternatively and/or additionally to `SSLCACertificatePath` for explicitly constructing the server certificate chain which is sent to the browser in addition to the server certificate. It is especially useful to avoid conflicts with CA certificates when using client authentication. Because although placing a CA certificate of the server certificate chain into `SSLCACertificatePath` has the same effect for the certificate chain construction, it has the side-effect that client certificates issued by this same CA certificate are also accepted on client authentication. That's usually not one expect.

But be careful: Providing the certificate chain works only if you are using a *single* (either RSA *or* DSA) based server certificate. If you are using a coupled RSA+DSA certificate pair, this will work only if actually both certificates use the *same* certificate chain. Else the browsers will be confused in this situation.

Example
```
SSLCertificateChainFile /usr/local/apache/conf/ssl.crt/ca.crt
```

SSLCertificateFile Directive

Description:	Server PEM-encoded X.509 Certificate file
Syntax:	`SSLCertificateFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive points to the PEM-encoded Certificate file for the server and optionally also to the corresponding RSA or DSA Private Key file for it (contained in the same file). If the contained Private Key is encrypted the Pass Phrase dialog is forced at startup time. This directive can be used up to two times (referencing different filenames) when both a RSA and a DSA based server certificate is used in parallel.

Example
```
SSLCertificateFile /usr/local/apache/conf/ssl.crt/server.crt
```

SSLCertificateKeyFile Directive

Description:	Server PEM-encoded Private Key file
Syntax:	`SSLCertificateKeyFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive points to the PEM-encoded Private Key file for the server. If the Private Key is not combined with the Certificate in the `SSLCertificateFile`, use this additional directive to point to the file with the stand-alone Private Key. When `SSLCertificateFile` is used and the file contains both the Certificate and the Private Key this directive need not be used. But we strongly discourage this practice. Instead we recommend you to separate the Certificate and the Private Key. If the contained Private Key is encrypted, the Pass Phrase dialog is forced at startup time. This directive can be used up to two times (referencing different filenames) when both a RSA and a DSA based private key is used in parallel.

Example
```
SSLCertificateKeyFile /usr/local/apache/conf/ssl.key/server.key
```

SSLCipherSuite Directive

Description:	Cipher Suite available for negotiation in SSL handshake
Syntax:	`SSLCipherSuite` *`cipher-spec`*
Default:	`SSLCipherSuite ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP`
Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This complex directive uses a colon-separated *cipher-spec* string consisting of OpenSSL cipher specifications to configure the Cipher Suite the client is permitted to negotiate in the SSL handshake phase. Notice that this directive can be used both in per-server and per-directory context. In per-server context it applies to the standard SSL handshake when a connection is established. In per-directory context it forces a SSL renegotation with the reconfigured Cipher Suite after the HTTP request was read but before the HTTP response is sent.

An SSL cipher specification in *cipher-spec* is composed of 4 major attributes plus a few extra minor ones:

- *Key Exchange Algorithm*:
  RSA or Diffie-Hellman variants.
- *Authentication Algorithm*:
  RSA, Diffie-Hellman, DSS or none.
- *Cipher/Encryption Algorithm*:
  DES, Triple-DES, RC4, RC2, IDEA or none.
- *MAC Digest Algorithm*:
  MD5, SHA or SHA1.

An SSL cipher can also be an export cipher and is either a SSLv2 or SSLv3/TLSv1 cipher (here TLSv1 is equivalent to SSLv3). To specify which ciphers to use, one can either specify all the Ciphers, one at a time, or use aliases to specify the preference and order for the ciphers (see Table 1).

Tag	**Description**
Key Exchange Algorithm:	
`kRSA`	RSA key exchange
`kDHr`	Diffie-Hellman key exchange with RSA key
`kDHd`	Diffie-Hellman key exchange with DSA key
`kEDH`	Ephemeral (temp.key) Diffie-Hellman key exchange (no cert)
Authentication Algorithm:	
`aNULL`	No authentication
`aRSA`	RSA authentication
`aDSS`	DSS authentication
`aDH`	Diffie-Hellman authentication
Cipher Encoding Algorithm:	
`eNULL`	No encoding

Tag	**Description**
`DES`	DES encoding
`3DES`	Triple-DES encoding
`RC4`	RC4 encoding
`RC2`	RC2 encoding
`IDEA`	IDEA encoding
MAC Digest Algorithm:	
`MD5`	MD5 hash function
`SHA1`	SHA1 hash function
`SHA`	SHA hash function
Aliases:	
`SSLv2`	all SSL version 2.0 ciphers
`SSLv3`	all SSL version 3.0 ciphers
`TLSv1`	all TLS version 1.0 ciphers
`EXP`	all export ciphers
`EXPORT40`	all 40-bit export ciphers only
`EXPORT56`	all 56-bit export ciphers only
`LOW`	all low strength ciphers (no export, single DES)
`MEDIUM`	all ciphers with 128 bit encryption
`HIGH`	all ciphers using Triple-DES
`RSA`	all ciphers using RSA key exchange
`DH`	all ciphers using Diffie-Hellman key exchange
`EDH`	all ciphers using Ephemeral Diffie-Hellman key exchange
`ADH`	all ciphers using Anonymous Diffie-Hellman key exchange
`DSS`	all ciphers using DSS authentication
`NULL`	all ciphers using no encryption

Now where this becomes interesting is that these can be put together to specify the order and ciphers you wish to use. To speed this up there are also aliases (`SSLv2, SSLv3, TLSv1, EXP, LOW, MEDIUM, HIGH`) for certain groups of ciphers. These tags can be joined together with prefixes to form the *cipher-spec*. Available prefixes are:

- none: add cipher to list
- `+`: add ciphers to list and pull them to current location in list
- `-`: remove cipher from list (can be added later again)
- `!`: kill cipher from list completely (can **not** be added later again)

A simpler way to look at all of this is to use the ``` ``openssl ciphers -v" ``` command which provides a nice way to successively create the correct *cipher-spec* string. The default *cipher-spec* string is ``` ``ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP" ``` which means the following: first, remove from consideration any ciphers that do not authenticate, i.e. for SSL only the Anonymous Diffie-Hellman ciphers. Next, use ciphers using RC4 and RSA. Next include the high, medium and then the low security ciphers. Finally *pull* all SSLv2 and export ciphers to the end of the list.

```
$ openssl ciphers -v 'ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP'
NULL-SHA                SSLv3 Kx=RSA      Au=RSA  Enc=None      Mac=SHA1
NULL-MD5                SSLv3 Kx=RSA      Au=RSA  Enc=None      Mac=MD5
EDH-RSA-DES-CBC3-SHA    SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
...                     ...               ...     ...           ...
EXP-RC4-MD5             SSLv3 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
EXP-RC2-CBC-MD5         SSLv2 Kx=RSA(512) Au=RSA  Enc=RC2(40)   Mac=MD5  export
EXP-RC4-MD5             SSLv2 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
```

The complete list of particular RSA & DH ciphers for SSL is given in Table 2.

Example

```
SSLCipherSuite RSA:!EXP:!NULL:+HIGH:+MEDIUM:-LOW
```

Cipher-Tag	**Protocol**	**Key Ex.**	**Auth.**	**Enc.**	**MAC**	**Type**
RSA Ciphers:						
`DES-CBC3-SHA`	SSLv3	RSA	RSA	3DES(168)	SHA1	
`DES-CBC3-MD5`	SSLv2	RSA	RSA	3DES(168)	MD5	
`IDEA-CBC-SHA`	SSLv3	RSA	RSA	IDEA(128)	SHA1	
`RC4-SHA`	SSLv3	RSA	RSA	RC4(128)	SHA1	
`RC4-MD5`	SSLv3	RSA	RSA	RC4(128)	MD5	
`IDEA-CBC-MD5`	SSLv2	RSA	RSA	IDEA(128)	MD5	
`RC2-CBC-MD5`	SSLv2	RSA	RSA	RC2(128)	MD5	
`RC4-MD5`	SSLv2	RSA	RSA	RC4(128)	MD5	
`DES-CBC-SHA`	SSLv3	RSA	RSA	DES(56)	SHA1	
`RC4-64-MD5`	SSLv2	RSA	RSA	RC4(64)	MD5	
`DES-CBC-MD5`	SSLv2	RSA	RSA	DES(56)	MD5	
`EXP-DES-CBC-SHA`	SSLv3	RSA(512)	RSA	DES(40)	SHA1	export
`EXP-RC2-CBC-MD5`	SSLv3	RSA(512)	RSA	RC2(40)	MD5	export
`EXP-RC4-MD5`	SSLv3	RSA(512)	RSA	RC4(40)	MD5	export
`EXP-RC2-CBC-MD5`	SSLv2	RSA(512)	RSA	RC2(40)	MD5	export
`EXP-RC4-MD5`	SSLv2	RSA(512)	RSA	RC4(40)	MD5	export
`NULL-SHA`	SSLv3	RSA	RSA	None	SHA1	
`NULL-MD5`	SSLv3	RSA	RSA	None	MD5	
Diffie-Hellman Ciphers:						
`ADH-DES-CBC3-SHA`	SSLv3	DH	None	3DES(168)	SHA1	
`ADH-DES-CBC-SHA`	SSLv3	DH	None	DES(56)	SHA1	
`ADH-RC4-MD5`	SSLv3	DH	None	RC4(128)	MD5	
`EDH-RSA-DES-CBC3-SHA`	SSLv3	DH	RSA	3DES(168)	SHA1	
`EDH-DSS-DES-CBC3-SHA`	SSLv3	DH	DSS	3DES(168)	SHA1	
`EDH-RSA-DES-CBC-SHA`	SSLv3	DH	RSA	DES(56)	SHA1	
`EDH-DSS-DES-CBC-SHA`	SSLv3	DH	DSS	DES(56)	SHA1	
`EXP-EDH-RSA-DES-CBC-SHA`	SSLv3	DH(512)	RSA	DES(40)	SHA1	export

Cipher-Tag	Protocol	Key Ex.	Auth.	Enc.	MAC	Type
EXP-EDH-DSS-DES-CBC-SHA	SSLv3	DH(512)	DSS	DES(40)	SHA1	export
EXP-ADH-DES-CBC-SHA	SSLv3	DH(512)	None	DES(40)	SHA1	export
EXP-ADH-RC4-MD5	SSLv3	DH(512)	None	RC4(40)	MD5	export

SSLEngine Directive

Description:	SSL Engine Operation Switch
Syntax:	`SSLEngine on|off`
Default:	`SSLEngine off`
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive toggles the usage of the SSL/TLS Protocol Engine. This is usually used inside a `<VirtualHost>` section to enable SSL/TLS for a particular virtual host. By default the SSL/TLS Protocol Engine is disabled for both the main server and all configured virtual hosts.

Example

```
<VirtualHost _default_:443>
SSLEngine on
...
</VirtualHost>
```

SSLMutex Directive

Description:	Semaphore for internal mutual exclusion of operations
Syntax:	`SSLMutex` *`type`*
Default:	`SSLMutex none`
Context:	server config
Status:	Extension
Module:	mod_ssl

This configures the SSL engine's semaphore (aka. lock) which is used for mutual exclusion of operations which have to be done in a synchronized way between the pre-forked Apache server processes. This directive can only be used in the global server context because it's only useful to have one global mutex.

The following Mutex *types* are available:

- `none`
  This is the default where no Mutex is used at all. Use it at your own risk. But because currently the Mutex is mainly used for synchronizing write access to the SSL Session Cache you can live without it as long as you accept a sometimes garbled Session Cache. So it's not recommended to leave this the default. Instead configure a real Mutex.
- `file:/path/to/mutex`
  This is the portable and (under Unix) always provided Mutex variant where a physical (lock-)file is used as the Mutex. Always use a local disk filesystem for `/path/to/mutex` and never a file

residing on a NFS- or AFS-filesystem. Note: Internally, the Process ID (PID) of the Apache parent process is automatically appended to `/path/to/mutex` to make it unique, so you don't have to worry about conflicts yourself. Notice that this type of mutex is not available under the Win32 environment. There you *have* to use the semaphore mutex.

- `sem`
  This is the most elegant but also most non-portable Mutex variant where a SysV IPC Semaphore (under Unix) and a Windows Mutex (under Win32) is used when possible. It is only available when the underlying platform supports it.

Example

```
SSLMutex file:/usr/local/apache/logs/ssl_mutex
```

SSLOptions Directive

Description:	Configure various SSL engine run-time options
Syntax:	`SSLOptions [+|-]`*`option`*` ...`
Context:	server config, virtual host, directory, .htaccess
Override:	Options
Status:	Extension
Module:	mod_ssl

This directive can be used to control various run-time options on a per-directory basis. Normally, if multiple `SSLOptions` could apply to a directory, then the most specific one is taken completely; the options are not merged. However if *all* the options on the `SSLOptions` directive are preceded by a plus (`+`) or minus (`-`) symbol, the options are merged. Any options preceded by a `+` are added to the options currently in force, and any options preceded by a `-` are removed from the options currently in force.

The available *option*s are:

- `StdEnvVars`
  When this option is enabled, the standard set of SSL related CGI/SSI environment variables are created. This per default is disabled for performance reasons, because the information extraction step is a rather expensive operation. So one usually enables this option for CGI and SSI requests only.
- `CompatEnvVars`
  When this option is enabled, additional CGI/SSI environment variables are created for backward compatibility to other Apache SSL solutions. Look in the Compatibility[3] chapter for details on the particular variables generated.
- `ExportCertData`
  When this option is enabled, additional CGI/SSI environment variables are created: `SSL_SERVER_CERT`, `SSL_CLIENT_CERT` and `SSL_CLIENT_CERT_CHAIN`*n* (with *n* = 0,1,2,..). These contain the PEM-encoded X.509 Certificates of server and client for the current HTTPS connection and can be used by CGI scripts for deeper Certificate checking. Additionally all other certificates of the client certificate chain are provided, too. This bloats up the environment a little bit which is why you have to use this option to enable it on demand.
- `FakeBasicAuth`
  When this option is enabled, the Subject Distinguished Name (DN) of the Client X509 Certificate is translated into a HTTP Basic Authorization username. This means that the standard

Apache authentication methods can be used for access control. The user name is just the Subject of the Client's X509 Certificate (can be determined by running OpenSSL's `openssl x509` command: `openssl x509 -noout -subject -in` *certificate*`.crt`). Note that no password is obtained from the user. Every entry in the user file needs this password: ``xxj31ZMTZzkVA'', which is the DES-encrypted version of the word `password''. Those who live under MD5-based encryption (for instance under FreeBSD or BSD/OS, etc.) should use the following MD5 hash of the same word: ``$1$OXLyS...$Owx8s2/m9/gfkcRVXzgoE/''.

- `StrictRequire`
  This *forces* forbidden access when `SSLRequireSSL` or `SSLRequire` successfully decided that access should be forbidden. Usually the default is that in the case where a ``Satisfy any'' directive is used, and other access restrictions are passed, denial of access due to `SSLRequireSSL` or `SSLRequire` is overridden (because that's how the Apache `Satisfy` mechanism should work.) But for strict access restriction you can use `SSLRequireSSL` and/or `SSLRequire` in combination with an ``SSLOptions +StrictRequire''. Then an additional ``Satisfy Any'' has no chance once mod_ssl has decided to deny access.
- `OptRenegotiate`
  This enables optimized SSL connection renegotiation handling when SSL directives are used in per-directory context. By default a strict scheme is enabled where *every* per-directory reconfiguration of SSL parameters causes a *full* SSL renegotiation handshake. When this option is used mod_ssl tries to avoid unnecessary handshakes by doing more granular (but still safe) parameter checks. Nevertheless these granular checks sometimes maybe not what the user expects, so enable this on a per-directory basis only, please.

Example

```
SSLOptions +FakeBasicAuth -StrictRequire
<Files ~ "\.(cgi|shtml)$">
SSLOptions +StdEnvVars +CompatEnvVars -ExportCertData
<Files>
```

SSLPassPhraseDialog Directive

Description:	Type of pass phrase dialog for encrypted private keys
Syntax:	`SSLPassPhraseDialog` *type*
Default:	`SSLPassPhraseDialog builtin`
Context:	server config
Status:	Extension
Module:	mod_ssl

When Apache starts up it has to read the various Certificate (see `SSLCertificateFile`) and Private Key (see `SSLCertificateKeyFile`) files of the SSL-enabled virtual servers. Because for security reasons the Private Key files are usually encrypted, mod_ssl needs to query the administrator for a Pass Phrase in order to decrypt those files. This query can be done in two ways which can be configured by *type*:

- `builtin`
  This is the default where an interactive terminal dialog occurs at startup time just before Apache detaches from the terminal. Here the administrator has to manually enter the Pass Phrase for each encrypted Private Key file. Because a lot of SSL-enabled virtual hosts can be configured, the following reuse-scheme is used to minimize the dialog: When a Private Key file is encrypted, all known Pass Phrases (at the beginning there are none, of course) are tried. If one of those known

Pass Phrases succeeds no dialog pops up for this particular Private Key file. If none succeeded, another Pass Phrase is queried on the terminal and remembered for the next round (where it perhaps can be reused).

This scheme allows mod_ssl to be maximally flexible (because for N encrypted Private Key files you *can* use N different Pass Phrases - but then you have to enter all of them, of course) while minimizing the terminal dialog (i.e. when you use a single Pass Phrase for all N Private Key files this Pass Phrase is queried only once).

- `exec:/path/to/program`
  Here an external program is configured which is called at startup for each encrypted Private Key file. It is called with two arguments (the first is of the form ``servername:portnumber'', the second is either ``RSA'' or ``DSA''), which indicate for which server and algorithm it has to print the corresponding Pass Phrase to `stdout`. The intent is that this external program first runs security checks to make sure that the system is not compromised by an attacker, and only when these checks were passed successfully it provides the Pass Phrase.

  Both these security checks, and the way the Pass Phrase is determined, can be as complex as you like. Mod_ssl just defines the interface: an executable program which provides the Pass Phrase on `stdout`. Nothing more or less! So, if you're really paranoid about security, here is your interface. Anything else has to be left as an exercise to the administrator, because local security requirements are so different.

  The reuse-algorithm above is used here, too. In other words: The external program is called only once per unique Pass Phrase.

Example:

```
SSLPassPhraseDialog exec:/usr/local/apache/sbin/pp-filter
```

SSLProtocol Directive

Description:	Configure usable SSL protocol flavors
Syntax:	`SSLProtocol [+|-]`*`protocol`*` ...`
Default:	`SSLProtocol all`
Context:	server config, virtual host
Override:	Options
Status:	Extension
Module:	mod_ssl

This directive can be used to control the SSL protocol flavors mod_ssl should use when establishing its server environment. Clients then can only connect with one of the provided protocols.

The available (case-insensitive) *protocol*s are:

- `SSLv2`
  This is the Secure Sockets Layer (SSL) protocol, version 2.0. It is the original SSL protocol as designed by Netscape Corporation.
- `SSLv3`
  This is the Secure Sockets Layer (SSL) protocol, version 3.0. It is the successor to SSLv2 and the currently (as of February 1999) de-facto standardized SSL protocol from Netscape Corporation. It's supported by almost all popular browsers.

- `TLSv1`
  This is the Transport Layer Security (TLS) protocol, version 1.0. It is the successor to SSLv3 and currently (as of February 1999) still under construction by the Internet Engineering Task Force (IETF). It's still not supported by any popular browsers.
- `All`
  This is a shortcut for ``` ``+SSLv2 +SSLv3 +TLSv1" ``` and a convinient way for enabling all protocols except one when used in combination with the minus sign on a protocol as the example above shows.

Example

```
# enable SSLv3 and TLSv1, but not SSLv2
SSLProtocol all -SSLv2
```

SSLProxyCACertificateFile Directive

Description:	File of concatenated PEM-encoded CA Certificates for Remote Server Auth
Syntax:	`SSLProxyCACertificateFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the *all-in-one* file where you can assemble the Certificates of Certification Authorities (CA) whose *remote servers* you deal with. These are used for Remote Server Authentication. Such a file is simply the concatenation of the various PEM-encoded Certificate files, in order of preference. This can be used alternatively and/or additionally to `SSLProxyCACertificatePath`.

Example

```
SSLProxyCACertificateFile
/usr/local/apache/conf/ssl.crt/ca-bundle-remote-server.crt
```

SSLProxyCACertificatePath Directive

Description:	Directory of PEM-encoded CA Certificates for Remote Server Auth
Syntax:	`SSLProxyCACertificatePath` *`directory-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the directory where you keep the Certificates of Certification Authorities (CAs) whose remote servers you deal with. These are used to verify the remote server certificate on Remote Server Authentication.

The files in this directory have to be PEM-encoded and are accessed through hash filenames. So usually you can't just place the Certificate files there: you also have to create symbolic links named *hash-value*`.N`. And you should always make sure this directory contains the appropriate symbolic links. Use the `Makefile` which comes with mod_ssl to accomplish this task.

Example
```
SSLProxyCACertificatePath /usr/local/apache/conf/ssl.crt/
```

SSLProxyCARevocationFile Directive

Description:	File of concatenated PEM-encoded CA CRLs for Remote Server Auth
Syntax:	`SSLProxyCARevocationFile` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the *all-in-one* file where you can assemble the Certificate Revocation Lists (CRL) of Certification Authorities (CA) whose *remote servers* you deal with. These are used for Remote Server Authentication. Such a file is simply the concatenation of the various PEM-encoded CRL files, in order of preference. This can be used alternatively and/or additionally to `SSLProxyCARevocationPath`.

Example
```
SSLProxyCARevocationFile
/usr/local/apache/conf/ssl.crl/ca-bundle-remote-server.crl
```

SSLProxyCARevocationPath Directive

Description:	Directory of PEM-encoded CA CRLs for Remote Server Auth
Syntax:	`SSLProxyCARevocationPath` *`directory-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the directory where you keep the Certificate Revocation Lists (CRL) of Certification Authorities (CAs) whose remote servers you deal with. These are used to revoke the remote server certificate on Remote Server Authentication.

The files in this directory have to be PEM-encoded and are accessed through hash filenames. So usually you have not only to place the CRL files there. Additionally you have to create symbolic links named *hash-value*`.rN`. And you should always make sure this directory contains the appropriate symbolic links. Use the `Makefile` which comes with `mod_ssl` to accomplish this task.

Example
```
SSLProxyCARevocationPath /usr/local/apache/conf/ssl.crl/
```

SSLProxyCipherSuite Directive

Description:	Cipher Suite available for negotiation in SSL proxy handshake
Syntax:	`SSLProxyCipherSuite` *`cipher-spec`*
Default:	`SSLProxyCipherSuite`

	`ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP`
Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

Equivalent to `SSLCipherSuite`, but for the proxy connection. Please refer to `SSLCipherSuite` for additional information.

SSLProxyEngine Directive

Description:	SSL Proxy Engine Operation Switch
Syntax:	`SSLProxyEngine on|off`
Default:	`SSLProxyEngine off`
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive toggles the usage of the SSL/TLS Protocol Engine for proxy. This is usually used inside a `<VirtualHost>` section to enable SSL/TLS for proxy usage in a particular virtual host. By default the SSL/TLS Protocol Engine is disabled for proxy image both for the main server and all configured virtual hosts.

Example
```
<VirtualHost _default_:443>
SSLProxyEngine on
...
</VirtualHost>
```

SSLProxyMachineCertificateFile Directive

Description:	File of concatenated PEM-encoded CA certificates for proxy server client certificates
Syntax:	`SSLProxyMachineCertificateFile` *`filename`*
Default:	`None`
Context:	server config
Override:	Not applicable
Status:	Extension
Module:	mod_ssl

This directive sets the all-in-one file where you keep the certificates of Certification Authorities (CAs) whose proxy client certificates are used for authentication of the proxy server to remote servers.

This referenced file is simply the concatenation of the various PEM-encoded certificate files, in order of preference. Use this directive alternatively or additionally to `SSLProxyMachineCertificatePath`.

Example:

```
SSLProxyMachineCertificatePath /usr/local/apache/conf/ssl.crt/
```

SSLProxyMachineCertificatePath Directive

Description:	Directory of PEM-encoded CA certificates for proxy server client certificates
Syntax:	`SSLProxyMachineCertificatePath` *`directory`*
Default:	`None`
Context:	server config
Override:	Not applicable
Status:	Extension
Module:	mod_ssl

This directive sets the directory where you keep the certificates of Certification Authorities (CAs) whose proxy client certificates are used for authentication of the proxy server to remote servers.

The files in this directory must be PEM-encoded and are accessed through hash filenames. Additionally, you must create symbolic links named *`hash-value`*`.N`. And you should always make sure this directory contains the appropriate symbolic links. Use the Makefile which comes with mod_ssl to accomplish this task.

Example:

```
SSLProxyMachineCertificatePath /usr/local/apache/conf/ssl.crt/
```

SSLProxyProtocol Directive

Description:	Configure usable SSL protocol flavors for proxy usage
Syntax:	`SSLProxyProtocol [+|-]`*`protocol`*` ...`
Default:	`SSLProxyProtocol all`
Context:	server config, virtual host
Override:	Options
Status:	Extension
Module:	mod_ssl

This directive can be used to control the SSL protocol flavors mod_ssl should use when establishing its server environment for proxy . It will only connect to servers using one of the provided protocols.

Please refer to `SSLProtocol` for additional information.

SSLProxyVerify Directive

Description:	Type of remote server Certificate verification
Syntax:	`SSLProxyVerify` *`level`*
Default:	`SSLProxyVerify none`
Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig

Status:	Extension
Module:	mod_ssl

This directive sets the Certificate verification level for the remote server Authentication. Notice that this directive can be used both in per-server and per-directory context. In per-server context it applies to the remote server authentication process used in the standard SSL handshake when a connection is established. In per-directory context it forces a SSL renegotation with the reconfigured remote server verification level after the HTTP request was read but before the HTTP response is sent.

The following levels are available for *level*:

- **none**: no remote server Certificate is required at all
- **optional**: the remote server *may* present a valid Certificate
- **require**: the remote server *has to* present a valid Certificate
- **optional_no_ca**: the remote server may present a valid Certificate
  but it need not to be (successfully) verifiable.

In practice only levels **none** and **require** are really interesting, because level **optional** doesn't work with all servers and level **optional_no_ca** is actually against the idea of authentication (but can be used to establish SSL test pages, etc.)

Example
```
SSLProxyVerify require
```

SSLProxyVerifyDepth Directive

Description:	Maximum depth of CA Certificates in Remote Server Certificate verification
Syntax:	`SSLVerifyDepth` *`number`*
Default:	`SSLVerifyDepth 1`
Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This directive sets how deeply mod_ssl should verify before deciding that the remote server does not have a valid certificate. Notice that this directive can be used both in per-server and per-directory context. In per-server context it applies to the client authentication process used in the standard SSL handshake when a connection is established. In per-directory context it forces a SSL renegotation with the reconfigured remote server verification depth after the HTTP request was read but before the HTTP response is sent.

The depth actually is the maximum number of intermediate certificate issuers, i.e. the number of CA certificates which are max allowed to be followed while verifying the remote server certificate. A depth of 0 means that self-signed remote server certificates are accepted only, the default depth of 1 means the remote server certificate can be self-signed or has to be signed by a CA which is directly known to the server (i.e. the CA's certificate is under `SSLProxyCACertificatePath`), etc.

Example

```
SSLProxyVerifyDepth 10
```

SSLRandomSeed Directive

Description:	Pseudo Random Number Generator (PRNG) seeding source
Syntax:	`SSLRandomSeed` *`context source`* `[`*`bytes`*`]`
Context:	server config
Status:	Extension
Module:	mod_ssl

This configures one or more sources for seeding the Pseudo Random Number Generator (PRNG) in OpenSSL at startup time (*context* is `startup`) and/or just before a new SSL connection is established (*context* is `connect`). This directive can only be used in the global server context because the PRNG is a global facility.

The following *source* variants are available:

- `builtin`
  This is the always available builtin seeding source. It's usage consumes minimum CPU cycles under runtime and hence can be always used without drawbacks. The source used for seeding the PRNG contains of the current time, the current process id and (when applicable) a randomly choosen 1KB extract of the inter-process scoreboard structure of Apache. The drawback is that this is not really a strong source and at startup time (where the scoreboard is still not available) this source just produces a few bytes of entropy. So you should always, at least for the startup, use an additional seeding source.
- `file:/path/to/source`
  This variant uses an external file `/path/to/source` as the source for seeding the PRNG. When *bytes* is specified, only the first *bytes* number of bytes of the file form the entropy (and *bytes* is given to `/path/to/source` as the first argument). When *bytes* is not specified the whole file forms the entropy (and `0` is given to `/path/to/source` as the first argument). Use this especially at startup time, for instance with an available `/dev/random` and/or `/dev/urandom` devices (which usually exist on modern Unix derivates like FreeBSD and Linux).

  But be careful: Usually `/dev/random` provides only as much entropy data as it actually has, i.e. when you request 512 bytes of entropy, but the device currently has only 100 bytes available two things can happen: On some platforms you receive only the 100 bytes while on other platforms the read blocks until enough bytes are available (which can take a long time). Here using an existing `/dev/urandom` is better, because it never blocks and actually gives the amount of requested data. The drawback is just that the quality of the received data may not be the best.

  On some platforms like FreeBSD one can even control how the entropy is actually generated, i.e. by which system interrupts. More details one can find under *rndcontrol(8)* on those platforms. Alternatively, when your system lacks such a random device, you can use tool like EGD [5] (Entropy Gathering Daemon) and run it's client program with the `exec:/path/to/program/` variant (see below) or use `egd:/path/to/egd-socket` (see below).
- `exec:/path/to/program`
  This variant uses an external executable `/path/to/program` as the source for seeding the PRNG. When *bytes* is specified, only the first *bytes* number of bytes of its `stdout` contents form the entropy. When *bytes* is not specified, the entirety of the data produced on `stdout` form the entropy. Use this only at startup time when you need a very strong seeding with the help of an external program (for instance as in the example above with the `truerand` utility you can find in

the mod_ssl distribution which is based on the AT&T *truerand* library). Using this in the connection context slows down the server too dramatically, of course. So usually you should avoid using external programs in that context.

- `egd:/path/to/egd-socket` (Unix only)

  This variant uses the Unix domain socket of the external Entropy Gathering Daemon (EGD) (see http://www.lothar.com/tech /crypto/[5]) to seed the PRNG. Use this if no random device exists on your platform.

Example

```
SSLRandomSeed startup builtin
SSLRandomSeed startup file:/dev/random
SSLRandomSeed startup file:/dev/urandom 1024
SSLRandomSeed startup exec:/usr/local/bin/truerand 16
SSLRandomSeed connect builtin
SSLRandomSeed connect file:/dev/random
SSLRandomSeed connect file:/dev/urandom 1024
```

SSLRequire Directive

Description:	Allow access only when an arbitrarily complex boolean expression is true
Syntax:	`SSLRequire` *`expression`*
Context:	directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This directive specifies a general access requirement which has to be fulfilled in order to allow access. It's a very powerful directive because the requirement specification is an arbitrarily complex boolean expression containing any number of access checks.

The *expression* must match the following syntax (given as a BNF grammar notation):

```
expr     ::= "true" | "false"
           | "!" expr
           | expr "&&" expr
           | expr "||" expr
           | "(" expr ")"
           | comp

comp     ::= word "==" word | word "eq" word
           | word "!=" word | word "ne" word
           | word "<"  word | word "lt" word
           | word "<=" word | word "le" word
           | word ">"  word | word "gt" word
           | word ">=" word | word "ge" word
           | word "in" "{" wordlist "}"
           | word "=~" regex
           | word "!~" regex

wordlist ::= word
           | wordlist "," word

word     ::= digit
```

```
                | cstring
                | variable
                | function

     digit    ::= [0-9]+
     cstring  ::= "..."
     variable ::= "%{" varname "}"
     function ::= funcname "(" funcargs ")"
```

while for `varname` any variable from Table 3 can be used. Finally for `funcname` the following functions are available:

- `file(`*filename*`)`
  This function takes one string argument and expands to the contents of the file. This is especially useful for matching this contents against a regular expression, etc.

Notice that *expression* is first parsed into an internal machine representation and then evaluated in a second step. Actually, in Global and Per-Server Class context *expression* is parsed at startup time and at runtime only the machine representation is executed. For Per-Directory context this is different: here *expression* has to be parsed and immediately executed for every request.

Example

```
SSLRequire ( %{SSL_CIPHER} !~ m/^(EXP|NULL)-/ \
and %{SSL_CLIENT_S_DN_O} eq "Snake Oil, Ltd." \
and %{SSL_CLIENT_S_DN_OU} in {"Staff", "CA", "Dev"} \
and %{TIME_WDAY} >= 1 and %{TIME_WDAY} <= 5 \
and %{TIME_HOUR} >= 8 and %{TIME_HOUR} <= 20 ) \
or %{REMOTE_ADDR} =~ m/^192\.76\.162\.[0-9]+$/
```

Standard CGI/1.0 and Apache variables:

HTTP_USER_AGENT	PATH_INFO	AUTH_TYPE
HTTP_REFERER	QUERY_STRING	SERVER_SOFTWARE
HTTP_COOKIE	REMOTE_HOST	API_VERSION
HTTP_FORWARDED	REMOTE_IDENT	TIME_YEAR
HTTP_HOST	IS_SUBREQ	TIME_MON
HTTP_PROXY_CONNECTION	DOCUMENT_ROOT	TIME_DAY
HTTP_ACCEPT	SERVER_ADMIN	TIME_HOUR
HTTP:headername	SERVER_NAME	TIME_MIN
THE_REQUEST	SERVER_PORT	TIME_SEC
REQUEST_METHOD	SERVER_PROTOCOL	TIME_WDAY
REQUEST_SCHEME	REMOTE_ADDR	TIME
REQUEST_URI	REMOTE_USER	ENV:**variablename**
REQUEST_FILENAME		

SSL-related variables:

HTTPS	SSL_CLIENT_M_VERSION	SSL_SERVER_M_VERSION
	SSL_CLIENT_M_SERIAL	SSL_SERVER_M_SERIAL
SSL_PROTOCOL	SSL_CLIENT_V_START	SSL_SERVER_V_START
SSL_SESSION_ID	SSL_CLIENT_V_END	SSL_SERVER_V_END
SSL_CIPHER	SSL_CLIENT_S_DN	SSL_SERVER_S_DN
SSL_CIPHER_EXPORT	SSL_CLIENT_S_DN_C	SSL_SERVER_S_DN_C
SSL_CIPHER_ALGKEYSIZE	SSL_CLIENT_S_DN_ST	SSL_SERVER_S_DN_ST
SSL_CIPHER_USEKEYSIZE	SSL_CLIENT_S_DN_L	SSL_SERVER_S_DN_L
SSL_VERSION_LIBRARY	SSL_CLIENT_S_DN_O	SSL_SERVER_S_DN_O
SSL_VERSION_INTERFACE	SSL_CLIENT_S_DN_OU	SSL_SERVER_S_DN_OU
	SSL_CLIENT_S_DN_CN	SSL_SERVER_S_DN_CN

	SSL_CLIENT_S_DN_T	SSL_SERVER_S_DN_T
	SSL_CLIENT_S_DN_I	SSL_SERVER_S_DN_I
	SSL_CLIENT_S_DN_G	SSL_SERVER_S_DN_G
	SSL_CLIENT_S_DN_S	SSL_SERVER_S_DN_S
	SSL_CLIENT_S_DN_D	SSL_SERVER_S_DN_D
	SSL_CLIENT_S_DN_UID	SSL_SERVER_S_DN_UID
	SSL_CLIENT_S_DN_Email	SSL_SERVER_S_DN_Email
	SSL_CLIENT_I_DN	SSL_SERVER_I_DN
	SSL_CLIENT_I_DN_C	SSL_SERVER_I_DN_C
	SSL_CLIENT_I_DN_ST	SSL_SERVER_I_DN_ST
	SSL_CLIENT_I_DN_L	SSL_SERVER_I_DN_L
	SSL_CLIENT_I_DN_O	SSL_SERVER_I_DN_O
	SSL_CLIENT_I_DN_OU	SSL_SERVER_I_DN_OU
	SSL_CLIENT_I_DN_CN	SSL_SERVER_I_DN_CN
	SSL_CLIENT_I_DN_T	SSL_SERVER_I_DN_T
	SSL_CLIENT_I_DN_I	SSL_SERVER_I_DN_I
	SSL_CLIENT_I_DN_G	SSL_SERVER_I_DN_G
	SSL_CLIENT_I_DN_S	SSL_SERVER_I_DN_S
	SSL_CLIENT_I_DN_D	SSL_SERVER_I_DN_D
	SSL_CLIENT_I_DN_UID	SSL_SERVER_I_DN_UID
	SSL_CLIENT_I_DN_Email	SSL_SERVER_I_DN_Email
	SSL_CLIENT_A_SIG	SSL_SERVER_A_SIG
	SSL_CLIENT_A_KEY	SSL_SERVER_A_KEY
	SSL_CLIENT_CERT	SSL_SERVER_CERT
	SSL_CLIENT_CERT_CHAIN**n**	
	SSL_CLIENT_VERIFY	

SSLRequireSSL Directive

Description:	Deny access when SSL is not used for the HTTP request
Syntax:	`SSLRequireSSL`
Context:	directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This directive forbids access unless HTTP over SSL (i.e. HTTPS) is enabled for the current connection. This is very handy inside the SSL-enabled virtual host or directories for defending against configuration errors that expose stuff that should be protected. When this directive is present all requests are denied which are not using SSL.

Example

```
SSLRequireSSL
```

SSLSessionCache Directive

Description:	Type of the global/inter-process SSL Session Cache
Syntax:	`SSLSessionCache` *type*
Default:	`SSLSessionCache none`
Context:	server config
Status:	Extension
Module:	mod_ssl

This configures the storage type of the global/inter-process SSL Session Cache. This cache is an optional facility which speeds up parallel request processing. For requests to the same server process (via HTTP keep-alive), OpenSSL already caches the SSL session information locally. But because modern clients request inlined images and other data via parallel requests (usually up to four parallel requests are common) those requests are served by *different* pre-forked server processes. Here an inter-process cache helps to avoid unneccessary session handshakes.

The following two storage *type*s are currently supported:

- `none`
  This is the default and just disables the global/inter-process Session Cache. There is no drawback in functionality, but a noticeable speed penalty can be observed.
- `dbm:/path/to/datafile`
  This makes use of a DBM hashfile on the local disk to synchronize the local OpenSSL memory caches of the server processes. The slight increase in I/O on the server results in a visible request speedup for your clients, so this type of storage is generally recommended.
- `shm:/path/to/datafile[(`*size*`)]`

  This makes use of a high-performance hash table (approx. *size* bytes in size) inside a shared memory segment in RAM (established via `/path/to/datafile`) to synchronize the local OpenSSL memory caches of the server processes. This storage type is not available on all platforms.

Examples

```
SSLSessionCache dbm:/usr/local/apache/logs/ssl_gcache_data
SSLSessionCache shm:/usr/local/apache/logs/ssl_gcache_data(512000)
```

SSLSessionCacheTimeout Directive

Description:	Number of seconds before an SSL session expires in the Session Cache
Syntax:	`SSLSessionCacheTimeout` *seconds*
Default:	`SSLSessionCacheTimeout 300`
Context:	server config, virtual host
Status:	Extension
Module:	mod_ssl

This directive sets the timeout in seconds for the information stored in the global/inter-process SSL Session Cache and the OpenSSL internal memory cache. It can be set as low as 15 for testing, but should be set to higher values like 300 in real life.

Example

```
SSLSessionCacheTimeout 600
```

SSLVerifyClient Directive

Description:	Type of Client Certificate verification
Syntax:	`SSLVerifyClient` *level*
Default:	`SSLVerifyClient none`

Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This directive sets the Certificate verification level for the Client Authentication. Notice that this directive can be used both in per-server and per-directory context. In per-server context it applies to the client authentication process used in the standard SSL handshake when a connection is established. In per-directory context it forces a SSL renegotation with the reconfigured client verification level after the HTTP request was read but before the HTTP response is sent.

The following levels are available for *level*:

- **none**: no client Certificate is required at all
- **optional**: the client *may* present a valid Certificate
- **require**: the client *has to* present a valid Certificate
- **optional_no_ca**: the client may present a valid Certificate
  but it need not to be (successfully) verifiable.

In practice only levels **none** and **require** are really interesting, because level **optional** doesn't work with all browsers and level **optional_no_ca** is actually against the idea of authentication (but can be used to establish SSL test pages, etc.)

Example
```
SSLVerifyClient require
```

SSLVerifyDepth Directive

Description:	Maximum depth of CA Certificates in Client Certificate verification
Syntax:	`SSLVerifyDepth` *`number`*
Default:	`SSLVerifyDepth 1`
Context:	server config, virtual host, directory, .htaccess
Override:	AuthConfig
Status:	Extension
Module:	mod_ssl

This directive sets how deeply mod_ssl should verify before deciding that the clients don't have a valid certificate. Notice that this directive can be used both in per-server and per-directory context. In per-server context it applies to the client authentication process used in the standard SSL handshake when a connection is established. In per-directory context it forces a SSL renegotation with the reconfigured client verification depth after the HTTP request was read but before the HTTP response is sent.

The depth actually is the maximum number of intermediate certificate issuers, i.e. the number of CA certificates which are max allowed to be followed while verifying the client certificate. A depth of 0 means that self-signed client certificates are accepted only, the default depth of 1 means the client certificate can be self-signed or has to be signed by a CA which is directly known to the server (i.e. the CA's certificate is under `SSLCACertificatePath`), etc.

Example

```
SSLVerifyDepth 10
```

URI References

[1] http://www.openssl.org/
[2] http://httpd.apache.org/docs-2.1/ssl/
[3] http://httpd.apache.org/docs-2.1/ssl/ssl_compat.html
[4] http://httpd.apache.org/docs-2.1/mod/mod_log_config.html#formats
[5] http://www.lothar.com/tech/crypto/