## NAME

SSL_get_ex_data_X509_STORE_CTX_idx, SSL_CTX_set_verify, SSL_set_verify, SSL_CTX_set_verify_depth, SSL_set_verify_depth, SSL_verify_cb, SSL_verify_client_post_handshake, SSL_set_post_handshake_auth, SSL_CTX_set_post_handshake_auth - set various SSL/TLS parameters for peer certificate verification

## SYNOPSIS

```
#include <openssl/ssl.h>

typedef int (*SSL_verify_cb)(int preverify_ok, X509_STORE_CTX *x509_ctx);

void SSL_CTX_set_verify(SSL_CTX *ctx, int mode, SSL_verify_cb verify_callback);
void SSL_set_verify(SSL *ssl, int mode, SSL_verify_cb verify_callback);
SSL_get_ex_data_X509_STORE_CTX_idx(void);

void SSL_CTX_set_verify_depth(SSL_CTX *ctx, int depth);
void SSL_set_verify_depth(SSL *ssl, int depth);

int SSL_verify_client_post_handshake(SSL *ssl);
void SSL_CTX_set_post_handshake_auth(SSL_CTX *ctx, int val);
void SSL_set_post_handshake_auth(SSL *ssl, int val);
```

## DESCRIPTION

**SSL_CTX_set_verify()** sets the verification flags for **ctx** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**.

**SSL_set_verify()** sets the verification flags for **ssl** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**. In this case last **verify_callback** set specifically for this **ssl** remains. If no special **callback** was set before, the default callback for the underlying **ctx** is used, that was valid at the time **ssl** was created with **SSL_new**(3). Within the callback function, **SSL_get_ex_data_X509_STORE_CTX_idx** can be called to get the data index of the current SSL object that is doing the verification.

In client mode **verify_callback** may also call the **SSL_set_retry_verify**(3) function on the **SSL** object set in the *x509_store_ctx* ex data (see **SSL_get_ex_data_X509_STORE_CTX_idx**(3)) and return 1. This would be typically done in case the certificate verification was not yet able to succeed. This makes the handshake suspend and return control to the calling application with **SSL_ERROR_WANT_RETRY_VERIFY**. The application can for instance fetch further certificates or

cert status information needed for the verification. Calling **SSL_connect**(3) again resumes the connection attempt by retrying the server certificate verification step. This process may even be repeated if need be. Note that the handshake may still be aborted if a subsequent invocation of the callback (e.g., at a lower depth, or for a separate error condition) returns 0.

**SSL_CTX_set_verify_depth()** sets the maximum **depth** for the certificate chain verification that shall be allowed for **ctx**.

**SSL_set_verify_depth()** sets the maximum **depth** for the certificate chain verification that shall be allowed for **ssl**.

**SSL_CTX_set_post_handshake_auth()** and **SSL_set_post_handshake_auth()** enable the Post-Handshake Authentication extension to be added to the ClientHello such that post-handshake authentication can be requested by the server. If **val** is 0 then the extension is not sent, otherwise it is. By default the extension is not sent. A certificate callback will need to be set via **SSL_CTX_set_client_cert_cb()** if no certificate is provided at initialization.

**SSL_verify_client_post_handshake()** causes a CertificateRequest message to be sent by a server on the given **ssl** connection. The SSL_VERIFY_PEER flag must be set; the SSL_VERIFY_POST_HANDSHAKE flag is optional.

## NOTES

The verification of certificates can be controlled by a set of logically or'ed **mode** flags:

SSL_VERIFY_NONE

**Server mode:** the server will not send a client certificate request to the client, so the client will not send a certificate.

**Client mode:** if not using an anonymous cipher (by default disabled), the server will send a certificate which will be checked. The result of the certificate verification process can be checked after the TLS/SSL handshake using the **SSL_get_verify_result**(3) function. The handshake will be continued regardless of the verification result.

SSL_VERIFY_PEER

**Server mode:** the server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. The behaviour can be controlled by the additional SSL_VERIFY_FAIL_IF_NO_PEER_CERT, SSL_VERIFY_CLIENT_ONCE and SSL_VERIFY_POST_HANDSHAKE flags.

**Client mode:** the server certificate is verified. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. If no server certificate is sent, because an anonymous cipher is used, SSL_VERIFY_PEER is ignored.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

**Server mode:** if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a "handshake failure" alert. This flag must be used together with SSL_VERIFY_PEER.

**Client mode:** ignored (see BUGS)

SSL_VERIFY_CLIENT_ONCE

**Server mode:** only request a client certificate once during the connection. Do not ask for a client certificate again during renegotiation or post-authentication if a certificate was requested during the initial handshake. This flag must be used together with SSL_VERIFY_PEER.

**Client mode:** ignored (see BUGS)

SSL_VERIFY_POST_HANDSHAKE

**Server mode:** the server will not send a client certificate request during the initial handshake, but will send the request via **SSL_verify_client_post_handshake()**. This allows the SSL_CTX or SSL to be configured for post-handshake peer verification before the handshake occurs. This flag must be used together with SSL_VERIFY_PEER. TLSv1.3 only; no effect on pre-TLSv1.3 connections.

**Client mode:** ignored (see BUGS)

If the **mode** is SSL_VERIFY_NONE none of the other flags may be set.

The actual verification procedure is performed either using the built-in verification procedure or using another application provided verification function set with **SSL_CTX_set_cert_verify_callback**(3). The following descriptions apply in the case of the built-in procedure. An application provided procedure also has access to the verify depth information and the **verify_callback()** function, but the way this information is used may be different.

**SSL_CTX_set_verify_depth()** and **SSL_set_verify_depth()** set a limit on the number of certificates between the end-entity and trust-anchor certificates. Neither the end-entity nor the trust-anchor certificates count against **depth**. If the certificate chain needed to reach a trusted issuer is longer than **depth+2**, X509_V_ERR_CERT_CHAIN_TOO_LONG will be issued. The depth count is "level 0:peer

certificate", "level 1: CA certificate", "level 2: higher level CA certificate", and so on. Setting the maximum depth to 2 allows the levels 0, 1, 2 and 3 (0 being the end-entity and 3 the trust-anchor). The default depth limit is 100, allowing for the peer certificate, at most 100 intermediate CA certificates and a final trust anchor certificate.

The **verify_callback** function is used to control the behaviour when the SSL_VERIFY_PEER flag is set. It must be supplied by the application and receives two arguments: **preverify_ok** indicates, whether the verification of the certificate in question was passed (preverify_ok=1) or not (preverify_ok=0). **x509_ctx** is a pointer to the complete context used for the certificate chain verification.

The certificate chain is checked starting with the deepest nesting level (the root CA certificate) and worked upward to the peer's certificate. At each level signatures and issuer attributes are checked. Whenever a verification error is found, the error number is stored in **x509_ctx** and **verify_callback** is called with **preverify_ok**=0. By applying X509_CTX_store_* functions **verify_callback** can locate the certificate in question and perform additional steps (see EXAMPLES). If no error is found for a certificate, **verify_callback** is called with **preverify_ok**=1 before advancing to the next level.

The return value of **verify_callback** controls the strategy of the further verification process. If **verify_callback** returns 0, the verification process is immediately stopped with "verification failed" state. If SSL_VERIFY_PEER is set, a verification failure alert is sent to the peer and the TLS/SSL handshake is terminated. If **verify_callback** returns 1, the verification process is continued. If **verify_callback** always returns 1, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. The calling process can however retrieve the error code of the last verification error using **SSL_get_verify_result**(3) or by maintaining its own error storage managed by **verify_callback**.

If no **verify_callback** is specified, the default callback will be used. Its return value is identical to **preverify_ok**, so that any verification failure will lead to a termination of the TLS/SSL handshake with an alert message, if SSL_VERIFY_PEER is set.

After calling **SSL_set_post_handshake_auth()**, the client will need to add a certificate or certificate callback to its configuration before it can successfully authenticate. This must be called before **SSL_connect()**.

**SSL_verify_client_post_handshake()** requires that verify flags have been previously set, and that a client sent the post-handshake authentication extension. When the client returns a certificate the verify callback will be invoked. A write operation must take place for the Certificate Request to be sent to the client, this can be done with **SSL_do_handshake()** or **SSL_write_ex()**. Only one certificate request may be outstanding at any time.

When post-handshake authentication occurs, a refreshed NewSessionTicket message is sent to the client.

## BUGS

In client mode, it is not checked whether the SSL_VERIFY_PEER flag is set, but whether any flags other than SSL_VERIFY_NONE are set. This can lead to unexpected behaviour if SSL_VERIFY_PEER and other flags are not used as required.

## RETURN VALUES

The SSL*_set_verify*() functions do not provide diagnostic information.

The **SSL_verify_client_post_handshake()** function returns 1 if the request succeeded, and 0 if the request failed. The error stack can be examined to determine the failure reason.

## EXAMPLES

The following code sequence realizes an example **verify_callback** function that will always continue the TLS/SSL handshake regardless of verification failure, if wished. The callback realizes a verification depth limit with more informational output.

All verification errors are printed; information about the certificate chain is printed on request. The example is realized for a server that does allow but not require client certificates.

The example makes use of the ex_data technique to store application data into/retrieve application data from the SSL structure (see **CRYPTO_get_ex_new_index**(3), **SSL_get_ex_data_X509_STORE_CTX_idx**(3)).

```
...
typedef struct {
  int verbose_mode;
  int verify_depth;
  int always_continue;
} mydata_t;
int mydata_index;

...
static int verify_callback(int preverify_ok, X509_STORE_CTX *ctx)
{
    char    buf[256];
    X509   *err_cert;
    int     err, depth;
```

```
    SSL    *ssl;
    mydata_t *mydata;

    err_cert = X509_STORE_CTX_get_current_cert(ctx);
    err = X509_STORE_CTX_get_error(ctx);
    depth = X509_STORE_CTX_get_error_depth(ctx);

    /*
     * Retrieve the pointer to the SSL of the connection currently treated
     * and the application specific data stored into the SSL object.
     */
    ssl = X509_STORE_CTX_get_ex_data(ctx, SSL_get_ex_data_X509_STORE_CTX_idx());
    mydata = SSL_get_ex_data(ssl, mydata_index);

    X509_NAME_oneline(X509_get_subject_name(err_cert), buf, 256);

    /*
     * Catch a too long certificate chain. The depth limit set using
     * SSL_CTX_set_verify_depth() is by purpose set to "limit+1" so
     * that whenever the "depth>verify_depth" condition is met, we
     * have violated the limit and want to log this error condition.
     * We must do it here, because the CHAIN_TOO_LONG error would not
     * be found explicitly; only errors introduced by cutting off the
     * additional certificates would be logged.
     */
    if (depth > mydata->verify_depth) {
        preverify_ok = 0;
        err = X509_V_ERR_CERT_CHAIN_TOO_LONG;
        X509_STORE_CTX_set_error(ctx, err);
    }
    if (!preverify_ok) {
        printf("verify error:num=%d:%s:depth=%d:%s\n", err,
               X509_verify_cert_error_string(err), depth, buf);
    } else if (mydata->verbose_mode) {
        printf("depth=%d:%s\n", depth, buf);
    }

    /*
     * At this point, err contains the last verification error. We can use
     * it for something special
```

```
     */
    if (!preverify_ok && (err == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT)) {
        X509_NAME_oneline(X509_get_issuer_name(err_cert), buf, 256);
        printf("issuer= %s\n", buf);
    }

    if (mydata->always_continue)
        return 1;
    else
        return preverify_ok;
}
...

mydata_t mydata;

...
mydata_index = SSL_get_ex_new_index(0, "mydata index", NULL, NULL, NULL);

...
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | SSL_VERIFY_CLIENT_ONCE,
                   verify_callback);

/*
 * Let the verify_callback catch the verify_depth error so that we get
 * an appropriate error in the logfile.
 */
SSL_CTX_set_verify_depth(verify_depth + 1);

/*
 * Set up the SSL specific data into "mydata" and store it into th SSL
 * structure.
 */
mydata.verify_depth = verify_depth; ...
SSL_set_ex_data(ssl, mydata_index, &mydata);

...
SSL_accept(ssl);       /* check of success left out for clarity */
if (peer = SSL_get_peer_certificate(ssl)) {
    if (SSL_get_verify_result(ssl) == X509_V_OK) {
        /* The client sent a certificate which verified OK */
```

```
    }
}
```

## SEE ALSO

**ssl**(7), **SSL_new**(3), **SSL_CTX_get_verify_mode**(3), **SSL_get_verify_result**(3), **SSL_CTX_load_verify_locations**(3), **SSL_get_peer_certificate**(3), **SSL_CTX_set_cert_verify_callback**(3), **SSL_get_ex_data_X509_STORE_CTX_idx**(3), **SSL_CTX_set_client_cert_cb**(3), **CRYPTO_get_ex_new_index**(3)

## HISTORY

The SSL_VERIFY_POST_HANDSHAKE option, and the **SSL_verify_client_post_handshake()** and **SSL_set_post_handshake_auth()** functions were added in OpenSSL 1.1.1.

## COPYRIGHT