## NAME

SSL_CTX_dane_enable, SSL_CTX_dane_mtype_set, SSL_dane_enable, SSL_dane_tlsa_add, SSL_get0_dane_authority, SSL_get0_dane_tlsa, SSL_CTX_dane_set_flags, SSL_CTX_dane_clear_flags, SSL_dane_set_flags, SSL_dane_clear_flags - enable DANE TLS authentication of the remote TLS server in the local TLS client

## SYNOPSIS

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

int SSL_CTX_dane_enable(SSL_CTX *ctx);
int SSL_CTX_dane_mtype_set(SSL_CTX *ctx, const EVP_MD *md,
                           uint8_t mtype, uint8_t ord);
int SSL_dane_enable(SSL *s, const char *basedomain);
int SSL_dane_tlsa_add(SSL *s, uint8_t usage, uint8_t selector,
                      uint8_t mtype, const unsigned char *data, size_t dlen);
int SSL_get0_dane_authority(SSL *s, X509 **mcert, EVP_PKEY **mspki);
int SSL_get0_dane_tlsa(SSL *s, uint8_t *usage, uint8_t *selector,
                       uint8_t *mtype, const unsigned char **data,
                       size_t *dlen);
unsigned long SSL_CTX_dane_set_flags(SSL_CTX *ctx, unsigned long flags);
unsigned long SSL_CTX_dane_clear_flags(SSL_CTX *ctx, unsigned long flags);
unsigned long SSL_dane_set_flags(SSL *ssl, unsigned long flags);
unsigned long SSL_dane_clear_flags(SSL *ssl, unsigned long flags);
```

## DESCRIPTION

These functions implement support for DANE TLSA (RFC6698 and RFC7671) peer authentication.

**SSL_CTX_dane_enable()** must be called first to initialize the shared state required for DANE support. Individual connections associated with the context can then enable per-connection DANE support as appropriate. DANE authentication is implemented in the **X509_verify_cert**(3) function, and applications that override **X509_verify_cert**(3) via **SSL_CTX_set_cert_verify_callback**(3) are responsible to authenticate the peer chain in whatever manner they see fit.

**SSL_CTX_dane_mtype_set()** may then be called zero or more times to adjust the supported digest algorithms. This must be done before any SSL handles are created for the context.

The **mtype** argument specifies a DANE TLSA matching type and the **md** argument specifies the associated digest algorithm handle. The **ord** argument specifies a strength ordinal. Algorithms with a larger strength ordinal are considered more secure. Strength ordinals are used to implement RFC7671 digest algorithm agility. Specifying a **NULL** digest algorithm for a matching type disables support for

that matching type. Matching type **Full**(0) cannot be modified or disabled.

By default, matching type "SHA2-256(1)" (see RFC7218 for definitions of the DANE TLSA parameter acronyms) is mapped to "EVP_sha256()" with a strength ordinal of 1 and matching type "SHA2-512(2)" is mapped to "EVP_sha512()" with a strength ordinal of 2.

**SSL_dane_enable()** must be called before the SSL handshake is initiated with **SSL_connect**(3) if (and only if) you want to enable DANE for that connection. (The connection must be associated with a DANE-enabled SSL context). The **basedomain** argument specifies the RFC7671 TLSA base domain, which will be the primary peer reference identifier for certificate name checks. Additional server names can be specified via **SSL_add1_host**(3). The **basedomain** is used as the default SNI hint if none has yet been specified via **SSL_set_tlsext_host_name**(3).

**SSL_dane_tlsa_add()** may then be called one or more times, to load each of the TLSA records that apply to the remote TLS peer. (This too must be done prior to the beginning of the SSL handshake). The arguments specify the fields of the TLSA record. The **data** field is provided in binary (wire RDATA) form, not the hexadecimal ASCII presentation form, with an explicit length passed via **dlen**. The library takes a copy of the **data** buffer contents and the caller may free the original **data** buffer when convenient. A return value of 0 indicates that "unusable" TLSA records (with invalid or unsupported parameters) were provided. A negative return value indicates an internal error in processing the record.

The caller is expected to check the return value of each **SSL_dane_tlsa_add()** call and take appropriate action if none are usable or an internal error is encountered in processing some records.

If no TLSA records are added successfully, DANE authentication is not enabled, and authentication will be based on any configured traditional trust-anchors; authentication success in this case does not mean that the peer was DANE-authenticated.

**SSL_get0_dane_authority()** can be used to get more detailed information about the matched DANE trust-anchor after successful connection completion. The return value is negative if DANE verification failed (or was not enabled), 0 if an EE TLSA record directly matched the leaf certificate, or a positive number indicating the depth at which a TA record matched an issuer certificate. The complete verified chain can be retrieved via **SSL_get0_verified_chain**(3). The return value is an index into this verified chain, rather than the list of certificates sent by the peer as returned by **SSL_get_peer_cert_chain**(3).

If the **mcert** argument is not **NULL** and a TLSA record matched a chain certificate, a pointer to the matching certificate is returned via **mcert**. The returned address is a short-term internal reference to the certificate and must not be freed by the application. Applications that want to retain access to the certificate can call **X509_up_ref**(3) to obtain a long-term reference which must then be freed via

**X509_free**(3) once no longer needed.

If no TLSA records directly matched any elements of the certificate chain, but a **DANE-TA**(2) **SPKI**(1) **Full**(0) record provided the public key that signed an element of the chain, then that key is returned via **mspki** argument (if not NULL). In this case the return value is the depth of the top-most element of the validated certificate chain. As with **mcert** this is a short-term internal reference, and **EVP_PKEY_up_ref**(3) and **EVP_PKEY_free**(3) can be used to acquire and release long-term references respectively.

**SSL_get0_dane_tlsa()** can be used to retrieve the fields of the TLSA record that matched the peer certificate chain. The return value indicates the match depth or failure to match just as with **SSL_get0_dane_authority()**. When the return value is nonnegative, the storage pointed to by the **usage**, **selector**, **mtype** and **data** parameters is updated to the corresponding TLSA record fields. The **data** field is in binary wire form, and is therefore not NUL-terminated, its length is returned via the **dlen** parameter. If any of these parameters is NULL, the corresponding field is not returned. The **data** parameter is set to a short-term internal-copy of the associated data field and must not be freed by the application. Applications that need long-term access to this field need to copy the content.

**SSL_CTX_dane_set_flags()** and **SSL_dane_set_flags()** can be used to enable optional DANE verification features. **SSL_CTX_dane_clear_flags()** and **SSL_dane_clear_flags()** can be used to disable the same features. The **flags** argument is a bit-mask of the features to enable or disable. The **flags** set for an **SSL_CTX** context are copied to each **SSL** handle associated with that context at the time the handle is created. Subsequent changes in the context's **flags** have no effect on the **flags** set for the handle.

At present, the only available option is **DANE_FLAG_NO_DANE_EE_NAMECHECKS** which can be used to disable server name checks when authenticating via **DANE-EE**(3) TLSA records. For some applications, primarily web browsers, it is not safe to disable name checks due to "unknown key share" attacks, in which a malicious server can convince a client that a connection to a victim server is instead a secure connection to the malicious server. The malicious server may then be able to violate cross-origin scripting restrictions. Thus, despite the text of RFC7671, name checks are by default enabled for **DANE-EE**(3) TLSA records, and can be disabled in applications where it is safe to do so. In particular, SMTP and XMPP clients should set this option as SRV and MX records already make it possible for a remote domain to redirect client connections to any server of its choice, and in any case SMTP and XMPP clients do not execute scripts downloaded from remote servers.

## RETURN VALUES

The functions **SSL_CTX_dane_enable()**, **SSL_CTX_dane_mtype_set()**, **SSL_dane_enable()** and **SSL_dane_tlsa_add()** return a positive value on success. Negative return values indicate resource problems (out of memory, etc.) in the SSL library, while a return value of **0** indicates incorrect usage or

invalid input, such as an unsupported TLSA record certificate usage, selector or matching type. Invalid input also includes malformed data, either a digest length that does not match the digest algorithm, or a Full(0) (binary ASN.1 DER form) certificate or a public key that fails to parse.

The functions **SSL_get0_dane_authority()** and **SSL_get0_dane_tlsa()** return a negative value when DANE authentication failed or was not enabled, a nonnegative value indicates the chain depth at which the TLSA record matched a chain certificate, or the depth of the top-most certificate, when the TLSA record is a full public key that is its signer.

The functions **SSL_CTX_dane_set_flags()**, **SSL_CTX_dane_clear_flags()**, **SSL_dane_set_flags()** and **SSL_dane_clear_flags()** return the **flags** in effect before they were called.

## EXAMPLES

Suppose "smtp.example.com" is the MX host of the domain "example.com", and has DNSSEC-validated TLSA records. The calls below will perform DANE authentication and arrange to match either the MX hostname or the destination domain name in the SMTP server certificate. Wildcards are supported, but must match the entire label. The actual name matched in the certificate (which might be a wildcard) is retrieved, and must be copied by the application if it is to be retained beyond the lifetime of the SSL connection.

```
SSL_CTX *ctx;
SSL *ssl;
int (*verify_cb)(int ok, X509_STORE_CTX *sctx) = NULL;
int num_usable = 0;
const char *nexthop_domain = "example.com";
const char *dane_tlsa_domain = "smtp.example.com";
uint8_t usage, selector, mtype;

if ((ctx = SSL_CTX_new(TLS_client_method())) == NULL)
    /* error */
if (SSL_CTX_dane_enable(ctx) <= 0)
    /* error */
if ((ssl = SSL_new(ctx)) == NULL)
    /* error */
if (SSL_dane_enable(ssl, dane_tlsa_domain) <= 0)
    /* error */

/*
 * For many applications it is safe to skip DANE-EE(3) namechecks.  Do not
 * disable the checks unless "unknown key share" attacks pose no risk for
```

```
 * your application.
 */
SSL_dane_set_flags(ssl, DANE_FLAG_NO_DANE_EE_NAMECHECKS);

if (!SSL_add1_host(ssl, nexthop_domain))
    /* error */
SSL_set_hostflags(ssl, X509_CHECK_FLAG_NO_PARTIAL_WILDCARDS);

for (... each TLSA record ...) {
    unsigned char *data;
    size_t len;
    int ret;

    /* set usage, selector, mtype, data, len */

    /*
     * Opportunistic DANE TLS clients support only DANE-TA(2) or DANE-EE(3).
     * They treat all other certificate usages, and in particular PKIX-TA(0)
     * and PKIX-EE(1), as unusable.
     */
    switch (usage) {
    default:
    case 0:     /* PKIX-TA(0) */
    case 1:     /* PKIX-EE(1) */
        continue;
    case 2:     /* DANE-TA(2) */
    case 3:     /* DANE-EE(3) */
        break;
    }

    ret = SSL_dane_tlsa_add(ssl, usage, selector, mtype, data, len);
    /* free data as appropriate */

    if (ret < 0)
        /* handle SSL library internal error */
    else if (ret == 0)
        /* handle unusable TLSA record */
    else
        ++num_usable;
}
```

```
/*
 * At this point, the verification mode is still the default SSL_VERIFY_NONE.
 * Opportunistic DANE clients use unauthenticated TLS when all TLSA records
 * are unusable, so continue the handshake even if authentication fails.
 */
if (num_usable == 0) {
    /* Log all records unusable? */

    /* Optionally set verify_cb to a suitable non-NULL callback. */
    SSL_set_verify(ssl, SSL_VERIFY_NONE, verify_cb);
} else {
    /* At least one usable record.  We expect to verify the peer */

    /* Optionally set verify_cb to a suitable non-NULL callback. */

    /*
     * Below we elect to fail the handshake when peer verification fails.
     * Alternatively, use the permissive SSL_VERIFY_NONE verification mode,
     * complete the handshake, check the verification status, and if not
     * verified disconnect gracefully at the application layer, especially if
     * application protocol supports informing the server that authentication
     * failed.
     */
    SSL_set_verify(ssl, SSL_VERIFY_PEER, verify_cb);
}

/*
 * Load any saved session for resumption, making sure that the previous
 * session applied the same security and authentication requirements that
 * would be expected of a fresh connection.
 */

/* Perform SSL_connect() handshake and handle errors here */

if (SSL_session_reused(ssl)) {
    if (SSL_get_verify_result(ssl) == X509_V_OK) {
        /*
         * Resumed session was originally verified, this connection is
         * authenticated.
         */
```

```
    } else {
        /*
         * Resumed session was not originally verified, this connection is not
         * authenticated.
         */
    }
} else if (SSL_get_verify_result(ssl) == X509_V_OK) {
    const char *peername = SSL_get0_peername(ssl);
    EVP_PKEY *mspki = NULL;

    int depth = SSL_get0_dane_authority(ssl, NULL, &mspki);
    if (depth >= 0) {
        (void) SSL_get0_dane_tlsa(ssl, &usage, &selector, &mtype, NULL, NULL);
        printf("DANE TLSA %d %d %d %s at depth %d\n", usage, selector, mtype,
               (mspki != NULL) ? "TA public key verified certificate" :
               depth ? "matched TA certificate" : "matched EE certificate",
               depth);
    }
    if (peername != NULL) {
        /* Name checks were in scope and matched the peername */
        printf("Verified peername: %s\n", peername);
    }
} else {
    /*
     * Not authenticated, presumably all TLSA rrs unusable, but possibly a
     * callback suppressed connection termination despite the presence of
     * usable TLSA RRs none of which matched.  Do whatever is appropriate for
     * fresh unauthenticated connections.
     */
}
```

## NOTES

It is expected that the majority of clients employing DANE TLS will be doing "opportunistic DANE TLS" in the sense of RFC7672 and RFC7435. That is, they will use DANE authentication when DNSSEC-validated TLSA records are published for a given peer, and otherwise will use unauthenticated TLS or even cleartext.

Such applications should generally treat any TLSA records published by the peer with usages **PKIX-TA**(0) and **PKIX-EE**(1) as "unusable", and should not include them among the TLSA records used to authenticate peer connections. In addition, some TLSA records with supported usages may be

"unusable" as a result of invalid or unsupported parameters.

When a peer has TLSA records, but none are "usable", an opportunistic application must avoid cleartext, but cannot authenticate the peer, and so should generally proceed with an unauthenticated connection. Opportunistic applications need to note the return value of each call to **SSL_dane_tlsa_add()**, and if all return 0 (due to invalid or unsupported parameters) disable peer authentication by calling **SSL_set_verify**(3) with **mode** equal to **SSL_VERIFY_NONE**.

## SEE ALSO

**ssl**(7), **SSL_new**(3), **SSL_add1_host**(3), **SSL_set_hostflags**(3), **SSL_set_tlsext_host_name**(3), **SSL_set_verify**(3), **SSL_CTX_set_cert_verify_callback**(3), **SSL_get0_verified_chain**(3), **SSL_get_peer_cert_chain**(3), **SSL_get_verify_result**(3), **SSL_connect**(3), **SSL_get0_peername**(3), **X509_verify_cert**(3), **X509_up_ref**(3), **X509_free**(3), **EVP_get_digestbyname**(3), **EVP_PKEY_up_ref**(3), **EVP_PKEY_free**(3)

## HISTORY

These functions were added in OpenSSL 1.1.0.

## COPYRIGHT