NAME

AnyEvent::TLS − SSLv2/SSLv3/TLSv1 contexts for use in AnyEvent::Handle

SYNOPSIS

```
# via AnyEvent::Handle

use AnyEvent;
use AnyEvent::Handle;
use AnyEvent::Socket;

# simple https-client
my $handle = new AnyEvent::Handle
   connect  => [$host, $port],
   tls      => "connect",
   tls_ctx  => { verify => 1, verify_peername => "https" },
   ...

# simple ssl-server
tcp_server undef, $port, sub {
   my ($fh) = @_;

   my $handle = new AnyEvent::Handle
      fh       => $fh,
      tls      => "accept",
      tls_ctx  => { cert_file => "my-server-keycert.pem" },
      ...

# directly

my $tls = new AnyEvent::TLS
   verify => 1,
   verify_peername => "ldaps",
   ca_file => "/etc/cacertificates.pem";
```

DESCRIPTION

This module is a helper module that implements TLS/SSL (Transport Layer Security/Secure Sockets Layer) contexts. A TLS context is a common set of configuration values for use in establishing TLS connections.

For some quick facts about SSL/TLS, see the section of the same name near the end of the document.

A single TLS context can be used for any number of TLS connections that wish to use the same certificates, policies etc.

Note that this module is inherently tied to Net::SSLeay, as this library is used to implement it. Since that perl module is rather ugly, and OpenSSL has a rather ugly license, AnyEvent might switch TLS providers at some future point, at which this API will change dramatically, at least in the Net::SSLeay−specific parts (most constructor arguments should still work, though).

Although this module does not require a specific version of Net::SSLeay, many features will gradually stop working, or bugs will be introduced with old versions (verification might succeed when it shouldn't − this is a real security issue). Version 1.35 is recommended, 1.33 should work, 1.32 might, and older versions are yours to keep.

USAGE EXAMPLES

See the AnyEvent::Handle manpage, NONFREQUENTLY ASKED QUESTIONS, for some actual usage examples.

PUBLIC METHODS AND FUNCTIONS

`$tls` = new AnyEvent::TLS key => value...

The constructor supports these arguments (all as key => value pairs).

method => "SSLv2" | "SSLv3" | "TLSv1" | "TLSv1_1" | "TLSv1_2" | "any"

The protocol parser to use. `SSLv2`, `SSLv3`, `TLSv1`, `TLSv1_1` and `TLSv1_2` will use a parser for those protocols only (so will *not* accept or create connections with/to other protocol versions), while `any` (the default) uses a parser capable of all three protocols.

The default is to use `"any"` but disable SSLv2. This has the effect of sending a SSLv2 hello, indicating the support for SSLv3 and TLSv1, but not actually negotiating an (insecure) SSLv2 connection.

Specifying a specific version is almost always wrong to use for a server speaking to a wide variety of clients (e.g. web browsers), and often wrong for a client. If you only want to allow a specific protocol version, use the `sslv2`, `sslv3`, `tlsv1`, `tlsv1_1` or `tlsv1_2` arguments instead.

For new services it is usually a good idea to enforce a `TLSv1` method from the beginning.

`TLSv1_1` and `TLSv1_2` require Net::SSLeay >= 1.55 and OpenSSL >= 1.0.1. Check the Net::SSLeay and OpenSSL documentations for more details.

sslv2 => `$enabled`

Enable or disable SSLv2 (normally *disabled*).

sslv3 => `$enabled`

Enable or disable SSLv3 (normally *enabled*).

tlsv1 => `$enabled`

Enable or disable TLSv1 (normally *enabled*).

tlsv1_1 => `$enabled`

Enable or disable TLSv1_1 (normally *enabled*).

This requires Net::SSLeay >= 1.55 and OpenSSL >= 1.0.1. Check the Net::SSLeay and OpenSSL documentations for more details.

tlsv1_2 => `$enabled`

Enable or disable TLSv1_2 (normally *enabled*).

This requires Net::SSLeay >= 1.55 and OpenSSL >= 1.0.1. Check the Net::SSLeay and OpenSSL documentations for more details.

verify => `$enable`

Enable or disable peer certificate checking (default is *disabled*, which is *not recommended*).

This is the "master switch" for all verify-related parameters and functions.

If it is disabled, then no peer certificate verification will be done – the connection will be encrypted, but the peer certificate won't be verified against any known CAs, or whether it is still valid or not. No peername verification or custom verification will be done either.

If enabled, then the peer certificate (required in client mode, optional in server mode, see `verify_require_client_cert`) will be checked against its CA certificate chain – that means there must be a signing chain from the peer certificate to any of the CA certificates you trust locally, as specified by the `ca_file` and/or `ca_path` and/or `ca_cert` parameters (or the system default CA repository, if all of those parameters are missing – see also the AnyEvent manpage for the description of PERL_ANYEVENT_CA_FILE).

Other basic checks, such as checking the validity period, will also be done, as well as optional peername/hostname/common name verification `verify_peername`.

An optional `verify_cb` callback can also be set, which will be invoked with the verification

results, and which can override the decision.

verify_require_client_cert => `$enable`
: Enable or disable mandatory client certificates (default is *disabled*). When this mode is enabled, then a client certificate will be required in server mode (a server certificate is mandatory, so in client mode, this switch has no effect).

verify_peername => `$scheme` | `$callback->`($tls, `$cert`, `$peername`)
: TLS only protects the data that is sent – it cannot automatically verify that you are really talking to the right peer. The reason is that certificates contain a ''common name'' (and a set of possible alternative ''names'') that need to be checked against the peername (usually, but not always, the DNS name of the server) in a protocol-dependent way.

  This can be implemented by specifying a callback that has to verify that the actual `$peername` matches the given certificate in `$cert`.

  Since this can be rather hard to implement, AnyEvent::TLS offers a variety of predefined ''schemes'' (lifted from IO::Socket::SSL) that are named like the protocols that use them:

  ldap (rfc4513), pop3,imap,acap (rfc2995), nntp (rfc4642)
  : Simple wildcards in subjectAltNames are possible, e.g. *.example.org matches www.example.org but not lala.www.example.org. If nothing from subjectAltNames matches, it checks against the common name, but there are no wildcards allowed.

  http (rfc2818)
  : Extended wildcards in subjectAltNames are possible, e.g. *.example.org or even www*.example.org. Wildcards in the common name are not allowed. The common name will be only checked if no host names are given in subjectAltNames.

  smtp (rfc3207)
  : This RFC isn't very useful in determining how to do verification so it just assumes that subjectAltNames are possible, but no wildcards are possible anywhere.

  [$wildcards_in_alt, `$wildcards_in_cn`, `$check_cn`]
  : You can also specify a scheme yourself by using an array reference with three integers.

    `$wildcards_in_alt` and `$wildcards_in_cn` specify whether and where wildcards (`*`) are allowed in subjectAltNames and the common name, respectively. `0` means no wildcards are allowed, `1` means they are allowed only as the first component (`*.example.org`), and `2` means they can be used anywhere (`www*.example.org`), except that very dangerous matches will not be allowed (`*.org` or `*`).

    `$check_cn` specifies if and how the common name field is checked: `0` means it will be completely ignored, `1` means it will only be used if no host names have been found in the subjectAltNames, and `2` means the common name will always be checked against the peername.

  You can specify either the name of the parent protocol (recommended, e.g. `http`, `ldap`), the protocol name as usually used in URIs (e.g. `https`, `ldaps`) or the RFC (not recommended, e.g. `rfc2995`, `rfc3920`).

  This verification will only be done when verification is enabled (`verify => 1`).

verify_cb => `$callback->`($tls, `$ref`, `$cn`, `$depth`, `$preverify_ok`, `$x509_store_ctx`, `$cert`)
: Provide a custom peer verification callback used by TLS sessions, which is called with the result of any other verification (`verify`, `verify_peername`).

  This callback will only be called when verification is enabled (`verify => 1`).

  `$tls` is the `AnyEvent::TLS` object associated with the session, while `$ref` is whatever the user associated with the session (usually an AnyEvent::Handle object when used by AnyEvent::Handle).

`$depth` is the current verification depth – `$depth = 0` means the certificate to verify is the peer certificate, higher levels are its CA certificate and so on. In most cases, you can just return `$preverify_ok` if the `$depth` is non-zero:

```
verify_cb => sub {
   my ($tls, $ref, $cn, $depth, $preverify_ok, $x509_store_ctx, $cert) =

   return $preverify_ok
      if $depth;

   # more verification
},
```

`$preverify_ok` is true iff the basic verification of the certificates was successful (a valid CA chain must exist, the certificate has passed basic validity checks, peername verification succeeded).

`$x509_store_ctx` is the Net::SSLeay::X509_CTX> object.

`$cert` is the `Net::SSLeay::X509` object representing the peer certificate, or zero if there was an error. You can call `AnyEvent::TLS::certname $cert` to get a nice user-readable string to identify the certificate.

The callback must return either `0` to indicate failure, or `1` to indicate success.

verify_client_once => `$enable`
: Enable or disable skipping the client certificate verification on renegotiations (default is *disabled*, the certificate will always be checked). Only makes sense in server mode.

ca_file => `$path`
: If this parameter is specified and non-empty, it will be the path to a file with (server) CA certificates in PEM format that will be loaded. Each certificate will look like:

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

You have to enable verify mode (`verify => 1`) for this parameter to have any effect.

ca_path => `$path`
: If this parameter is specified and non-empty, it will be the path to a directory with hashed CA certificate files in PEM format. When the ca certificate is being verified, the certificate will be hashed and looked up in that directory (see <http://www.openssl.org/docs/ssl/SSL_CTX_load_verify_locations.html> for details)

The certificates specified via `ca_file` take precedence over the ones found in `ca_path`.

You have to enable verify mode (`verify => 1`) for this parameter to have any effect.

ca_cert => `$string`
: In addition or instead of using `ca_file` and/or `ca_path`, you can also use `ca_cert` to directly specify the CA certificates (there can be multiple) in PEM format, in a string.

check_crl => `$enable`
: Enable or disable certificate revocation list checking. If enabled, then peer certificates will be checked against a list of revoked certificates issued by the CA. The revocation lists will be expected in the `ca_path` directory.

certificate verification will fail if this is enabled but no revocation list was found.

This requires OpenSSL >= 0.9.7b. Check the OpenSSL documentation for more details.

key_file => `$path`
: Path to the local private key file in PEM format (might be a combined certificate/private key file).

  The local certificate is used to authenticate against the peer − servers mandatorily need a certificate and key, clients can use a certificate and key optionally to authenticate, e.g. for log-in purposes.

  The key in the file should look similar this:

```
-----BEGIN RSA PRIVATE KEY-----
...header data
... (key data in base64 encoding) ...
-----END RSA PRIVATE KEY-----
```

key => `$string`
: The private key string in PEM format (see `key_file`, only one of `key_file` or `key` can be specified).

  The idea behind being able to specify a string is to avoid blocking in I/O. Unfortunately, Net::SSLeay fails to implement any interface to the needed OpenSSL functionality, this is currently implemented by writing to a temporary file.

cert_file => `$path`
: The path to the local certificate file in PEM format (might be a combined certificate/private key file, including chained certificates).

  The local certificate (and key) are used to authenticate against the peer − servers mandatorily need a certificate and key, clients can use certificate and key optionally to authenticate, e.g. for log-in purposes.

  The certificate in the file should look like this:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

  If the certificate file or string contain both the certificate and private key, then there is no need to specify a separate `key_file` or `key`.

  Additional signing certifiates to send to the peer (in SSLv3 and newer) can be specified by appending them to the certificate proper: the order must be from issuer certificate over any intermediate CA certificates to the root CA.

  So the recommended ordering for a combined key/cert/chain file, specified via `cert_file` or `cert` looks like this:

```
certificate private key
client/server certificate
ca 1, signing client/server certificate
ca 2, signing ca 1
...
```

cert => `$string`
: The local certificate in PEM format (might be a combined certificate/private key file). See `cert_file`.

  The idea behind being able to specify a string is to avoid blocking in I/O. Unfortunately, Net::SSLeay fails to implement any interface to the needed OpenSSL functionality, this is currently implemented by writing to a temporary file.

cert_password => `$string` | `$callback`->($tls)
: The certificate password − if the certificate is password-protected, then you can specify its password here.

Instead of providing a password directly (which is not so recommended), you can also provide a password-query callback. The callback will be called whenever a password is required to decode a local certificate, and is supposed to return the password.

dh_file => `$path`

Path to a file containing Diffie-Hellman parameters in PEM format, for use in servers. See also `dh` on how to specify them directly, or use a pre-generated set.

Diffie-Hellman key exchange generates temporary encryption keys that are not transferred over the connection, which means that even if the certificate key(s) are made public at a later time and a full dump of the connection exists, the key still cannot be deduced.

These ciphers are only available with SSLv3 and later (which is the default with AnyEvent::TLS), and are only used in server/accept mode. Anonymous DH protocols are usually disabled by default, and usually not even compiled into the underlying library, as they provide no direct protection against man-in-the-middle attacks. The same is true for the common practise of self-signed certificates that you have to accept first, of course.

dh => `$string`

Specify the Diffie-Hellman parameters in PEM format directly as a string (see `dh_file`), the default is `ffdhe3072` unless `dh_file` was specified.

AnyEvent::TLS supports supports a number of precomputed DH parameters, since computing them is expensive. They are:

```
# from RFC 7919 - recommended
ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, ffdhe8192

# from "Assigned Number for SKIP Protocols"
skip512, skip1024, skip2048, skip4096

# from schmorp
schmorp1024, schmorp1539, schmorp2048, schmorp4096, schmorp8192
```

It is said that 2048 bit DH parameters are safe till 2030, and DH parameters shorter than 900 bits are totally insecure.

To disable DH protocols completely, specify `undef` as `dh` parameter.

dh_single_use => `$enable`

Enables or disables ‘‘use only once’’ mode when using Diffie-Hellman key exchange. When enabled (default), each time a new key is exchanged a new Diffie-Hellman key is generated, which improves security as each key is only used once. When disabled, the key will be created as soon as the AnyEvent::TLS object is created and will be reused.

All the DH parameters supplied with AnyEvent::TLS should be safe with `dh_single_use` switched off, but YMMV.

cipher_list => `$string`

The list of ciphers to use, as a string (example: `AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH`). The format of this string and its default value is documented at <http://www.openssl.org/docs/apps/ciphers.html#CIPHER_STRINGS>.

session_ticket => `$enable`

Enables or disables RC5077 support (Session Resumption without Server-Side State). The default is disabled for clients, as many (buggy) TLS/SSL servers choke on it, but enabled for servers.

When enabled and supported by the server, a session ticket will be provided to the client, which allows fast resuming of connections.

prepare => `$coderef->`($tls)
If this argument is present, then it will be called with the new AnyEvent::TLS object after any other initialisation has bee done, in case you wish to fine-tune something...

`$tls` = new_from_ssleay AnyEvent::TLS `$ctx`
This constructor takes an existing Net::SSLeay SSL_CTX object (which is just an integer) and converts it into an `AnyEvent::TLS` object. This only works because AnyEvent::TLS is currently implemented using Net::SSLeay. As this is such a horrible perl module and OpenSSL has such an annoying license, this might change in the future, in which case this method might vanish.

`$ctx` = `$tls->`ctx
Returns the actual Net::SSLeay::CTX object (just an integer).

AnyEvent::TLS::init
AnyEvent::TLS does on-demand initialisation, and normally there is no need to call an initialise function.

As initialisation might take some time (to read e.g. `/dev/urandom`), this could be annoying in some highly interactive programs. In that case, you can call `AnyEvent::TLS::init` to make sure there will be no costly initialisation later. It is harmless to call `AnyEvent::TLS::init` multiple times.

`$certname` = AnyEvent::TLS::certname `$x509`
Utility function that returns a user-readable string identifying the X509 certificate object.

SSL/TLS QUICK FACTS

Here are some quick facts about TLS/SSL that might help you:

- A certificate is the public key part, a key is the private key part.

  While not strictly true, certificates are the things you can hand around publicly as a kind of identity, while keys should really be kept private, as proving that you have the private key is usually interpreted as being the entity behind the certificate.

- A certificate is signed by a CA (Certificate Authority).

  By signing, the CA basically claims that the certificate it signs really belongs to the identity named in it, verified according to the CA policies. For e.g. HTTPS, the CA usually makes some checks that the hostname mentioned in the certificate really belongs to the company/person that requested the signing and owns the domain.

- CAs can be certified by other CAs.

  Or by themselves – a certificate that is signed by a CA that is itself is called a self-signed certificate, a trust chain of length zero. When you find a certificate signed by another CA, which is in turn signed by another CA you trust, you have a trust chain of depth two.

- ''Trusting'' a CA means trusting all certificates it has signed.

  If you ''trust'' a CA certificate, then all certificates signed by it are automatically considered trusted as well.

- A successfully verified certificate means that you can be reasonably sure that whoever you are talking with really is who he claims he is.

  By verifying certificates against a number of CAs that you trust (meaning it is signed directly or indirectly by such a CA), you can find out that the other side really is whoever he claims, according to the CA policies, and your belief in the integrity of the CA.

- Verifying the certificate signature is not everything.

  Even when the certificate is correct, it might belong to somebody else: if www.attacker.com can make your computer believe that it is really called www.mybank.com (by making your DNS server believe this for example), then it could send you the certificate for www.attacker.com that your software trusts because it is signed by a CA you trust, and intercept all your traffic that you think goes to www.mybank.com. This works because your software sees that the certificate is correctly signed (for

www.attacker.com) and you think you are talking to your bank.

To thwart this attack vector, peername verification should be used, which basically checks that the certificate (for www.attacker.com) really belongs to the host you are trying to talk to (www.mybank.com), which in this example is not the case, as www.attacker.com (from the certificate) doesn't match www.mybank.com (the hostname used to create the connection).

So peername verification is almost as important as checking the CA signing. Unfortunately, every protocol implements this differently, if at all...

- Switching off verification is sometimes reasonable.

  You can switch off verification. You still get an encrypted connection that is protected against eavesdropping and injection – you just lose protection against man in the middle attacks, i.e. somebody else with enough abilities to intercept all traffic can masquerade herself as the other side.

  For many applications, switching off verification is entirely reasonable. Downloading random stuff from websites using HTTPS for no reason is such an application. Talking to your bank and entering TANs is not such an application.

- A SSL/TLS server always needs a certificate/key pair to operate, for clients this is optional.

  Apart from (usually disabled) anonymous cipher suites, a server always needs a certificate/key pair to operate.

  Clients almost never use certificates, but if they do, they can be used to authenticate the client, just as server certificates can be used to authenticate the server.

- SSL version 2 is very insecure.

  SSL version 2 is old and not only has it some security issues, SSLv2−only implementations are usually buggy, too, due to their age.

- Sometimes, even losing your ''private'' key might not expose all your data.

  With Diffie-Hellman ephemeral key exchange, you can lose the DH parameters (the ''keys''), but all your connections are still protected. Diffie-Hellman needs special set-up (done by default by AnyEvent::TLS).

SECURITY CONSIDERATIONS

When you use any of the options that pass in keys or certificates as strings (e.g. `ca_cert`), then, due to serious shortcomings in Net::SSLeay, this module creates a temporary file to store the string – see File::Temp and possibly its `safe_level` setting for more details on what to watch out for.

BUGS

Due to the abysmal code quality of Net::SSLeay, this module will leak small amounts of memory per TLS connection (currently at least one perl scalar).

AUTHORS

Marc Lehmann <schmorp@schmorp.de>.

Some of the API, documentation and implementation (verify_hostname), and a lot of ideas/workarounds/knowledge have been taken from the IO::Socket::SSL module. Care has been taken to keep the API similar to that and other modules, to the extent possible while providing a sensible API for AnyEvent.