Crypto Engines

A major component of TLS is of course the cryptography. The basic crypto requirements can be supplied from a variety of sources. In the current implementation we use the MIRACL core library, but there are many alternatives. To facilitate switching to a different crypto provider, all of the TLS crypto is now concentrated into a single module `tls_crypto_api`.

Some X.509 gotchas

X.509 is a mess, and everyone recognises that. Its elaborate ASN.1 encoding is a classic example of over-engineering a solution. And it can ambush you in various ways.

Take dates, as required for certificate expiry dates. There is an ASN.1 UTC encoding, which starts with the year – ‘2’, ‘0’, ‘2’, ‘1’. Except it doesn’t, it just encodes as ‘2’, ‘1’, with the assumption that the year is 20xx. Now one would hope that X.509 will be a relic in a computer museum somewhere by the time certificates are being issued which expire in 2099. But some-one obviously thinks it might last a bit longer than that, as there is an alternative ASN.1 GTM encoding, for “Generalized Time”. This is identical except that the ‘2’, ‘0’ is included, and, wait for it, expiry dates and times can now be encoded down to a fraction of a second. So your certificate can now expire at 21:15:14.375 in the year 3031. How useful is that?

Now almost everyone uses UTC. But in the list of root CAs there is one root certificate, just one, which uses the GTM time format. So it needs to be catered for. Just one small example of the madness that is X.509.

Pre-shared Keys

One of the nice features of TLS1.3 is that it can be used with an external pre-shared key, which can derived via alternative methods, and hence avoid using X.509 certificates and classic PKI for authentication. When such a pre-shared key (PSK) is agreed out-of-band it can be presented into TLS1.3 with a bespoke identifying label recognisable to both client and server, a KDF hash function for use by TLS (by default SHA256), and the PSK itself, typically a 128 or 256 bit random octet.

Again this can be tested using the `openssl s_server` utility, using the flags -`psk_identity` to specify the label, and `–psk` to specify the shared key.

Having fun with pre-shared keys

How about initially authenticating a client to a server and visa versa using a PAKE – a Password Authenticated Key Exchange? A PAKE protocol allows the two parties to agree a large high-entropy PSK from a low entropy shared value, like a humble 4-digit PIN.

Probably the best way to use any alternative means of authentication would be to integrate it directly into TLS, using the extension mechanism. This would keep the number of rounds in the overall key exchange to a minimum. However it also requires changes to TLS.

Instead a PAKE could be deployed to generate an agreed raw PSK, which could then be presented to TLS and processed as a standard external PSK. Following that initial bootstrapping connection we could use the TLS resumption mechanism to create a “ratchet” effect, using the ticketing mechanism. So an IoT device could be issued with an initial short secret that it shares with a server, which subsequently bootstraps us up into a Signal-like Diffie-Hellman ratchet for subsequent authenticated communications. After each connection, a new ticket is issued by the server to the client. We note that TLS1.3 tickets have a maximum lifetime of a week – so an IoT node would need to “check in” with its server at least once a week, if it wanted to maintain the ratchet. An advantage would be that tickets naturally expire after a week, which has some security advantages.