Embedded client implementation

I found this table on https://github.com/espressif/esp-wolfssl This refers to an implementation on an ESP32 embedded processor, the same one I will be experimenting with next.

Comparison of wolfSSL and mbedTLS

The following table shows a typical comparison between wolfSSL and mbedtls when `https_request` (which has server authentication) was run with both SSL/TLS libraries and with all respective configurations set to default. (mbedtls IN_CONTENT length and OUT_CONTENT length were set to 16384 bytes and 4096 bytes respectively)

Property	wolfSSL	mbedTLS
Total Heap Consumed	~19 Kb	~37 Kb
Task Stack Used	~2.2 Kb	~3.6 Kb
Bin size	~858 Kb	~736 Kb

The results are not at all surprising, and I suspect that our numbers will be very similar. The binary size is dominated by the requirement to include CA root certificates, and the Stack and Heap are dominated by the message sizes transmitted by the server which must be buffered by the client. Our code will of course use no heap memory. Currently it requires around 26Kbytes of stack memory, but I am confident that can be squeezed down further. But I suspect that it will end up somewhere between 16K and 32K, and that reduction below 16k is not really feasible.

I have refactored and simplified the Certificate Chain processing code, in particular have reduced its memory consumption to a minimum.

Keeping it small

One thing a client can do to defend itself is to ask for a smaller record fragment size in its clientHello. According to the RFC the server MAY respond to this request. However it seems that about half of the websites I tested ignore this request – which according to the RFC is permissible behaviour for the Server. Since the default record size is 16384 bytes, we will ask for this to be reduced to 4096 bytes. No harm in asking!

Since we are only using stack memory, we need to set some sensible maximum values. Some are quite easy to set – the memory required to process a root cert for example. There are 130 root certificates in the store I am using, and the largest is 2016 bytes long. The biggest memory hog is the buffer for accepting server messages, which must be big enough to take a full certificate chain. For now I am capping this at 8k bytes.

Forward secrecy

Apparently using this idea - https://eprint.iacr.org/2019/228.pdf – we can restore full-forward-secrecy for 0-RTT while sticking to the letter of the RFC, and only doing some new stuff on the server side. As it says “We show that our construction can immediately be used in TLS 1.3 0-RTT and deployed unilaterally by servers, without requiring any changes to clients or the protocol”. Seems like something we should consider.

Post quantum key exchange

Just realised how easy it is to go hybrid post-quantum, that is how to retain forward secrecy after a crypto-busting quantum computer is finally invented. First recall how TLS 1.3 works. The crypto starts with a simple elliptic curve based Diffie-Hellman key exchange which seeds all of the keys used subsequently. This key exchange is authenticated after-the-fact by the Server signing the whole transcript of the key exchange using the secret key associated with its public key embedded in its Certificate.

The point is that to achieve hybrid security, all we need to do is to replace the existing D-H key exchange with a post-quantum method. And unauthenticated key exchange is the simplest and most mature capability of all proposals for PQ security. So all we need to do is to replace the elliptic curve based key exchange with, for example, the NewHope lattice-based algorithm. Which is a very easy thing for us to do, as long as both client and server agree on it.